FULLSTACK WEB COMPONENTS

Complete Guide to Building UI Libraries with Web Components



newline

Stephen Belovarich

Fullstack Web Components

The Complete Guide to Building UI Libraries and Web Applications with Web Components

Written by Steve Belovarich
Edited by Nate Murray

© 2020 Fullstack.io

All rights reserved. No portion of the book manuscript may be reproduced, stored in a retrieval system, or transmitted in any form or by any means beyond the number of purchased copies, except for a single backup or archival copy. The code may be used freely in your projects, commercial or otherwise.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damagers in connection with or arising out of the use of the information or programs container herein.

Published by \newline



Contents

Book Revision 1
Join Our Discord 1
Bug Reports 1
Be notified of updates via Twitter 1
We'd love to hear from you! 1

Introduction **2**
Foreword 2
About the author 3
Web Components 4
About this book 5
Who is this book for 5
Introduction to Web Components 6
What are Web Components? 11
The many uses of Web Components 17
Implementing a design system with Web Components . 19
Conventions used in this Book 20
Storybook 22
Application 23
Cypress 24
Storyshots 24
Monorepo 24
Setting up the development environment 25
Prerequisites 25
Summary 29

Part One **32**
Part One 32
Specification 32
What you will learn 33
Developing in Part One 34

Getting to Know Web Components **35**
Chapter One 35
Introduction 35
What you will build 35
What you will learn 38
Acceptance criteria 38
Getting started 39
Scaffolding CardComponent 39
Structure content with template slots 45
Create controls for the story 49
Styling 52
Testing the card 60
Summary 72

Form-associated custom elements **74**
Chapter Two 74
Introduction 74
What you will build 76
What you will learn 79
Acceptance criteria 79
Getting started 80
Scaffolding TextInputComponent 80
Structure the content 82
Define the story 84
Form Associated and ElementInternals 86
Handling Validation 89
Listening for attribute changes 98
User feedback 112
Handling Disabled State 115

Make stories 116
Styling with CSS variables 119
Emulating HTMLInputElement 123
Making a form 129
Handling FormData 137
Testing TextInputComponent 139
Summary 144

Customized built-in elements 146
Chapter Three 146
Introduction 146
What you will build 149
What you will learn 150
Acceptance criteria 151
Getting started 151
Scaffolding ButtonComponent 151
Styling ButtonComponent 155
Making the form accessible 167
Testing ButtonComponent 179
Summary 181

Part Two 183
Part Two 183
UI Library 183
What you will learn 184
Developing in Part Two 184

Micro-library 186
Chapter Four 186
Introduction 186
What is a micro-library? 190
Coding in a monorepo 190
What you will build 192
What you will learn 192
Acceptance criteria 192
Getting started 193

Class Decorator 193
Method Decorator 227
Define the Listen decorator 231
Refactor TextInputComponent 238
Summary 241

Compound Custom Elements 243
Chapter Five 243
Component Communication 243
What you will build 247
What you will learn 247
Acceptance criteria 247
Getting started 248
Mocking Table Data in Storybook 248
Coding Compound Components 260
Attribute Drilling 260
BroadcastChannel 264
Rendering Columns and Rows 266
Styling TableComponent 272
Handling CRUD Events with BroadcastChannel 275
Making the Table Editable 280
Testing the Table 313
Summary 321

Powering Dialogs with HTML Templates 324
Chapter Six 324
HTML Templates 324
What you will build 328
What you will learn 328
Acceptance criteria 328
Getting started 329
Mocking the Dialog in Storybook 329
DialogComponent 337
Modal and Tooltip 343
Querying DOM Across Shadow Boundaries 350

DialogStack 362
Testing Dialogs 372
Summary 378

Distributing Custom Element Libraries 380
Chapter Seven 380
UI Libraries 380
What you will build 384
What you will learn 384
Acceptance criteria 385
Getting started 385
Distributing Custom Element UI Libraries 386
Providing Documentation with Custom Elements Manifest and Storybook 398
Set up Continuous Integration with Github Actions . . . 412
Summary 423

Part Three 425
Part Three 425
Application 425
What you will learn 425
Developing in Part Three 426

Application Development with Custom Elements 428
Chapter Eight 428
Application Development with Custom Elements 428
What you will build 430
What you will learn 431
Acceptance criteria 431
Getting started 432
Landing Page 432
App Header 436
Cookie Footer 447
Landing Page Content 460
Summary 467

Routing Custom Elements in a SPA **469**
Chapter Nine 469
Routing Custom Elements in a SPA 469
What you will build 472
What you will learn 472
Acceptance criteria 472
Getting started 472
Routing 473
Login View 483
Dashboard View 498
Summary 506

Server-side Rendering with Declarative Shadow DOM . . **508**
Chapter Ten 508
Server-Side Rendering Custom Elements 508
What you will build 516
What you will learn 517
Acceptance criteria 517
Getting started 517
Declarative Shadow DOM Templates 517
Shim 532
Ponyfill 536
Express Middleware 537
Summary 551

Fullstack Web Components **553**
Conclusion 553
Specification 554
Benefits 556
The future is bright 558

Changelog **560**
Revision 1 (04-04-2022) 560
Revision 2 (12-06-2022) 560

Book Revision

Revision 1 - Initial pre-release of the book - 04-04-2022

Revision 2 - Change book cover - 12-06-2022

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/webcomponents[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/webcomponents
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

Introduction

Acknowledgements

For my wife Susanne.

Foreword

This is a book about Web Components. UI components are a common pattern in many front end web development frameworks like Angular, React or Vue, but coding user interfaces in the context of a framework usually makes sharing those components difficult. Angular components can't be easily loaded in the context of React, for instance. The framework is a closed system.

If you can move past the framework, and possibly enjoy the simplification and standardization offered by web specifications, coding with Web Components can be quite rewarding. Working directly with specifications opens up a world once obscured by the framework. The web platform is yours to explore and find interesting ways to utilize parts of HTML, CSS, and JavaScript. By using web specifications to code UI components you can achieve extraordinary results. When used responsibly, adopting Web Components as your UI component pattern can lead to stellar performance, compliance with accessibility standards, and reusability.

At enterprise corporations it can be rather costly to support several development teams. One way to reduce operational overhead is to standardize UI components and reuse them across a software product. This cuts down on teams "reinventing the wheel". I've witnessed teams struggle with user interface development at several companies. It can be quite difficult to coordinate the use

of a UI library, particularly when it's coded with a JavaScript framework. User interfaces built with the Web Components are interoperable with any JavaScript framework. Dependencies become hard to maintain over a distributed software ecosystem, teams have conflicting priorities making it difficult to not just upgrade components, but also framework dependencies. When companies adopt Web Components as the solution for a UI component library, it can reduce the churn associated with framework dependencies. I wrote this guide for developing user interfaces using only web specifications because I believe it's in the best interests of software engineers at enterprise corporations to code in a way that allows user interfaces to be used over a long lifespan. Corporations shouldn't have the reinvest in user interfaces at the rate JavaScript frameworks go in and out of fashion.

It's one thing to help by providing examples or peer programming, but I didn't think that was enough. While coding a micro-library for developing Web Components, I found it difficult to find resources. This book is designed to take on a journey with Web Components as if you were developing a UI library at an enterprise corporation. You won't just get hands-on experience coding UI with Web Components, but also learn how to test and maintain those components in the context of a distributed UI library. This book will make you proficient enough to develop a UI library, web application or some other fantastical flights of fancy with Web Components.

About the author

Stephen Belovarich is a web engineer who started tinkering with web development in the 90s. After developing his first website, Stephen became enamored with the extraordinary potential of the web. For several years after attending college at Rensselaer Polytechnic Institute, he coded e-commerce solutions and marketing portals for freelance clients while creating net art for display in

fine art galleries. Specializing in user interface library and web application architecture, Stephen has developed interactive user experiences for Lexus, Ubiquiti Networks, NBC Universal, Nike, and Symantec. Stephen Belovarich is a Principal Software Development Engineer at Workday. Stephen is passionate about teaching the next generation of web engineers how to navigate the ever-expanding web platform. He taught computer science and engineering at General Assembly and Syracuse University, has presented at ngconf, ACT-W, and meetups around Portland, Oregon where he currently resides.

Web Components

Before the set of technologies known as Web Components were available engineers had to select an opinionated JavaScript framework to abstract user interfaces into "components". Web Components are interoperable with virtually any JavaScript library or framework because they are developed using only browser specifications. The lifespan of a user interface component built with Web Components can be much greater than a component built with a library since browser specifications exist for a very long time. Web Components can have dynamic templates, handle complex form behaviors, even inherit characteristics of familiar HTML elements like `<button>` or `<table>`. By coding a UI library with Web Components, you can ensure the components have a farther reach than components only compatible with a specific library or framework. Depending on implementation, components coded with Web Components specifications can out perform framework-based components as well, due to the lack of dependencies and overhead inherit in JavaScript frameworks.

About this book

In this book, we're going to learn how to develop a user interface library and web application with Web Components. We'll start by coding UI components using the set of specifications that comprise Web Components, increasing in complexity while covering real world problems: forms, custom templates, editable tables, and dialogs. In a lerna[4] monorepo, we'll code almost exclusively with Storybook[5] as our development environment and write tests for each component using Cypress. After configuring the UI library for distribution, we'll import the components into a fullstack web application built entirely with Web Components. In each chapter, you'll find step-by-step instructions for coding each component and developing a web application with Web Components. By the end of this book, you'll have developed a user interface library from a design system in Figma[6] and used those components in the context of a web application.

Who is this book for

This book is for anyone who wants to learn how to architect and develop a user interface library with Web Components. By learning how to architect reusable components, you'll be able to make better decisions when developing user interfaces in the future. Web Components are a fundamental building block of front end web development. Anyone with some prior knowledge of JavaScript, HTML, and CSS should be able to make sense of this book.

In the next section we're going to provide an in-depth introduction to Web Components and how they are used in the real world. This will get much of the theory and vocabulary out of the way and allow

[4] https://lerna.js.org
[5] https://storybook.js.org
[6] https://www.figma.com

us to buckle down and develop Web Components in the chapters that follow.

Introduction to Web Components

In the nineties web development was egalitarian. HTML provided a syntax anyone could learn. A `<p>` element displays a paragraph. An `<img>` element displays an image, provided a `src` attribute has a valid address for an image file. The HTML standard evolved over time, however the elements used in development in the nineties still work today.

What it takes to develop a website has dramatically changed over the years. Web development became even more sophisticated when corporations discovered an advantage web had over native: anyone can access software in a web browser and updates can be pushed server-side, without the need for the end-user to trigger a software update. JavaScript frameworks like React, Angular, and Vue filled in the gaps of the ECMAScript standard, giving web engineers sophisticated tools and patterns for coding feature-rich web applications. React, in particular, popularized the notion of classes with `React.createClass` before ECMAScript delivered the standards-based `class` with the ES2015 standard, otherwise known as ES6. While JavaScript frameworks created fragmentation with incompatible component models, JavaScript frameworks have one thing in common. React, Angular, and Vue all feature a discrete unit of functionality that encapsulates template, styling, and user interface behaviors called the "component".

Framework-based UI components solved a distinct problem in web development by providing a component model, but the web platform lagged behind. There was no way to create a custom HTML element. Web developers could only use the HTML elements provided by the HTML standard, and extending the prototype of a native HTML element was considered a bad practice. In web

application templates, each of the frameworks allow framework components to be declared alongside HTML elements, although components from two different frameworks don't necessarily work together in the same template. The opinionated implementation of one framework component clashes with another. Rendering multiple framework components in a single architecture can become extremely brittle. It's amazing how much JavaScript frameworks provide out of-the-box, but they don't really play nice together.

The fragmentation caused by JavaScript frameworks pose a problem at scale in corporations. Maintaining a consistent look and feel across a portfolio of many web applications is seemingly impossible when several engineering teams develop with different JavaScript frameworks. The same UI component could be duplicated many times over as the component is developed specifically for each framework. Pushing out changes to design and UX behavior becomes a herculean task, dealing the dependency chain of each JavaScript framework along the way.

Supporting accessible behaviors in framework-based UI components is challenging because when developing custom UI you essentially have to reinvent the wheel. Engineers may create a button component with a framework, but mistakingly use the template of a `<div>`, losing the accessibility behaviors of `<button>` in the process. The engineer would have to implement event listeners on keydown to listen for the spacebar and return keys being pressed, adding to the complexity of the codebase they maintain. Custom elements can extend from native HTML elements, retaining the accessibility behaviors of the element they extend from. This isn't to say Web Components aren't without challenges when it comes to accessibility. Shadow DOM can block screen readers from analyzing WAI-ARIA attributes when ids are deeply nested across Shadow boundaries, so developer intervention maybe needed to coax screen readers to obtain the desired results for accessibility.

Other problems with framework-based UI component development occur over time. Breaking changes in major versions of frameworks

can cause churn. As frameworks optimize over time, breaking changes to syntax or different ways of working force engineers to relearn how UI components are developed. Rarely nowadays would you see `React.createClass` in a modern React application, instead engineers would use the class-based `React.Component` or a functional implementation with React hooks. A standards-based Web Component has a longer lifespan than framework components. Like those HTML elements `<p>` and `<img>`, Web Components are part of the HTML specification. A custom element registered with the `CustomElementRegistry` will work in browsers for several years to come.

```
export class CardComponent extends HTMLElement {
  constructor() {
    super();
    const shadowRoot = this.attachShadow({ mode: 'open' }\
);
    const template = document.createElement('template');
    template.innerHTML = `
      <style>
                                :host {
                                        display: block;
                                        background: white;
                                        border-radius: 12px;
                                        border: 4px solid black;
          overflow: hidden;
          max-width: 320px;
          min-height: 120px;
                                }
      </style>
      <header>
        <slot name="header"></slot>
      </header>
      <section>
        <slot name="body"></slot>
```

```
      </section>
      <footer>
        <slot name="footer"></slot>
      </footer>
    `;
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
  }
}
customElements.define('in-card', CardComponent);
```

Web Components are interoperable, meaning custom elements can be used in the context JavaScript frameworks like any other HTML element.

```
export {CardComponent} from "./custom-element";

export const App = () => {
  return (
    <div>
      <in-card></in-card>
    </div>
  )
}

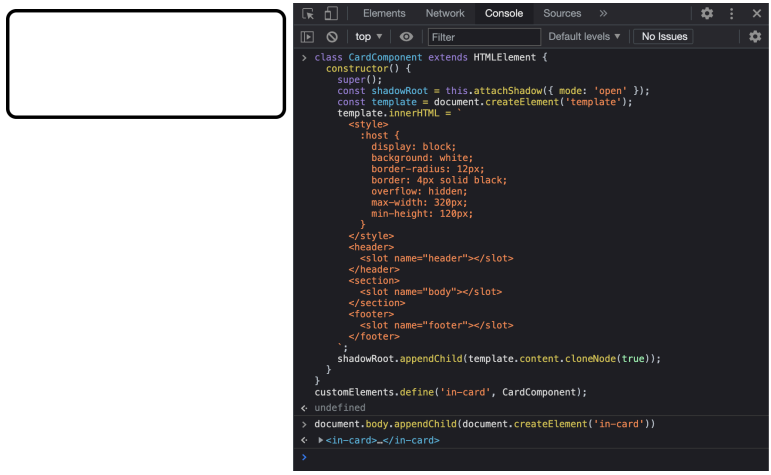
ReactDOM.render(<App />, document.querySelector("#root"));
```

Custom elements created with the Web Components standard can retain the accessibility behaviors of the HTML elements they extend. A customized built-in element that extends from `HTMLButtonElement` includes all the traits of a typical HTML `<button>` element. When the user presses Enter or Space, the button is activated without any custom logic. This isn't to say sometimes accessibility behaviors are not necessary to implement in autonomous custom elements or form-associated custom elements.

```
export class ButtonComponent extends HTMLButtonElement {
  constructor() {
    super();
  }
  connectedCallback() {
   this.addEventListener('click', this.onClick);
  }
  onClick() {
    this.dispatchEvent(new CustomEvent('buttonClicked'));
  }
}

customElements.define('in-button', ButtonComponent, {exte\
nds: 'button'});
```

Somehow we find ourselves at a place where web development is no longer egalitarian. The learning curve for web development is absurd for anyone who just wants to put a custom element on a page. Not only does someone have to learn the component model for a particular JavaScript framework, but how to test, maintain, and deploy complex framework code. Then another framework becomes popular and engineers have to relearn everything. We call this JavaScript fatigue. Part of the problem is the fragmentation, but in my opinion, it's mostly a problem with education. If people new to web development are skipping web standards in favor of learning frameworks, they are adding complexity, without learning about how things work. HTML elements are the most primitive Object we have to deal with in web development. Anyone can start developing UI with Web Components right in the browser.



While it's not practical to develop custom elements this way in the browser, the example above is useful to demonstrate that Web Components require zero dependencies to work.

What are Web Components?

Front end web development usually requires handling templates, styling, and logic that defines the behavior of discrete functional units, commonly referred to as UI components. Web Components are a set of browser specifications that provides encapsulation and organization for user-interface logic in a standard component model. The term "custom elements" is often used interchangeably with "web components", although custom elements are just one of the specifications that comprise Web Components.

Web Components is a marketing term that refers to several browser standards maintained by the **Web Hypertext Application Technology Working Group (WHATWG)** and the **World Wide Web Consortium (W3C)** in the ***HTML Living Standard*** and ***DOM Standard***, documents that establish the ongoing state of Hyper

Text Markup Language (HTML) and Document Object Model (DOM). WHATWG is comprised of web engineers from Apple, Google, Microsoft, Mozilla, Opera and the broader web development community. In 2019, W3C and WHATWG agreed to work together to maintain the HTML standard as a living document that constantly evolves over time.

In contrast to open source JavaScript frameworks that are maintained by corporate teams or the broader community, Web Components are developed through a rigorous standardization process. Once a particular Web Component standard is finalized, browser vendors prioritize and implement the specification. As of January 2020, all major web browsers including Apple Safari, Google Chrome, Microsoft Edge, Mozilla Firefox, and Opera implemented the major feature of ***Custom Elements v1*** specification: autonomous custom elements. Other specifications under the Web Components umbrella, such as Shadow DOM and HTML templates, have been part of the HTML standard for quite some time, while some like Declarative Shadow DOM or Constructable Stylesheets have yet to be included in all major web browsers. As HTML is a living document, support changes over time and any feature that lacks support in some browsers can be polyfilled. We'll look into this in more detail later. For now, let's define the various specifications that make up Web Components.

Custom elements

The cornerstone of the Web Components featureset, custom elements allow new HTML tags to be defined in a web browser. ***Custom Elements v1*** is the stable implementation of the custom elements specification. This standard allows you to define a new HTML element and the HTML element's JavaScript API. Other specifications like `ElementInternals` extend the functionality of custom elements beyond the set of specifications known as Custom Elements v1. Custom elements allow you to observe changes to

element attributes and manage content and styling for the new element's template. Custom elements behave like any other HTML element, with some syntactical sugar that promotes standardization across any custom element behavior.

There are three different types of custom elements: autonomous custom elements, customized built-in elements, and form-associated custom elements.

Autonomous custom elements

Autonomous custom elements allow you to define customized functionality for a **new HTML element**. Autonomous custom elements extend from the `HTMLElement` class. The most compatible custom element type, autonomous custom elements are available in all evergreen browsers. This specification is what people are typically referring to when describing Web Components or Custom Elements v1.

Customized built-in elements

Customized built-in elements **extend from known HTML elements**, retaining the behaviors of those elements. Customized built-in elements behave differently than autonomous custom elements, although this specification retains many of the features of autonomous custom elements. Customized built-in elements can be used in several evergreen browsers, except Apple Safari which requires a polyfill.

Form-associated custom elements

`HTMLFormElement`cannot access form controls like `HTMLInputElement` and `HTMLSelectElement` that are within a custom element's Shadow DOM tree. This means form validation is precarious at best. Form-associated custom elements provide an extended API from autonomous custom elements that gives you

hooks into form validation. This means you can still encapsulate a form control's template, while exposing the validity of the control to the parent `HTMLFormElement`. Form-associated custom elements open the door for the creation of truly customized user experiences that participate in HTML forms. Instead of merely relying on traditional form elements, it's now possible to make your own.

In the first part of this book, we'll investigate the similarities and differences between the different types of custom elements and see how they work together to provide complex, reusable components.

CustomElementRegistry

This interface on the `window.customElements` property provides methods for registering custom elements and querying registered elements. Every custom element must be registered with the `CustomElementRegistry` through a declarative syntax.

Shadow DOM

Shadow DOM is a browser technology that provides encapsulation for a HTML element, isolating the HTML element's template from the rest of DOM. When using Shadow DOM with custom elements, you can isolate `<style>` and `<script>` within the `ShadowRoot`[7]: the root node of a DOM substructure rendered outside the scope of DOM. The top level element that employs Shadow DOM is considered the `host` element. The `ShadowRoot` is attached to the `host` element. In contrast to Shadow DOM, anything outside of the shadow boundary in the global scope of the `document` is considered Light DOM.

When you query for an element on the document using `document.querySelector` or similar method, you are essentially querying the entire document because the Document Object Model

[7] https://developer.mozilla.org/en-US/docs/Web/API/ShadowRoot

(DOM) has a global scope. If you wanted to style an element or enhance an element with some logic via JavaScript, you would have to inject a `<style>` or `<script>` tag into the `document`, usually the `<head>` or before the closing `<body>` tag in the case of JavaScript that relies on querying for DOM that is fully loaded. Because of the global nature of DOM, CSS or JavaScript intended for one HTML element could collide with another the styling or logic wasn't intended for. When developing UI components it's extremely helpful to rely on an isolated scope for the component, so CSS and JavaScript collisions can't happen.

HTML Template

The `<template>` tag, first introduced in HTML5, allows HTML to be predefined and reused throughout DOM. HTML templates are parsed but not rendered until cloned. This behavior allows for custom element templates to be rendered performantly when HTML templates are used in conjunction with Shadow DOM.

HTML, CSS, and JSON Modules

Replacing the deprecated specification for JSON, CSS and HTML modules allow for custom elements to request resources asynchronously. Yet to be finalized the JSON, CSS and HTML modules specifications are currently in development by WHATWG and W3C.

CSS Shadow Parts

When Shadow DOM was introduced, the only way to style a custom element's `ShadowRoot` from the context of Light DOM was to use CSS variables. CSS variables were the only method that pierced the Shadow boundary.

CSS Shadow Parts is a CSS pseudo selector `::part` that allows styling of any elements within a Shadow DOM tree that has a matching `part` attribute. `::part` is available in all evergreen browsers. WHATWG is currently working on new specifications that enable greater flexibility when styling Shadow DOM including a specification known as Constructable Stylesheets.

Declarative Shadow DOM

Declarative Shadow DOM allows you declare Shadow DOM directly in HTML, without the need for JavaScript. This specification opens the doors for server-side rendering Shadow DOM and reduces the complexity when testing Web Components. With Declarative Shadow DOM, a custom element's Shadow Root tree can be statically generated and analyzed.

Typically there are limitations when server-side rendering custom elements. With the available tools, you may only be able to render a Shadow Root that is directly exposed to Light DOM, but not render any templates deeper in that Shadow Root's tree. Even though you've the ability for web crawlers to analyze Shadow DOM that's server-side rendered, there could be an unsettling flash of unstyled content or layout shifting while loading a Shadow Root stylesheet in the browser.

Declarative Shadow DOM negates prior limitations to server-side rendering custom elements by providing a way to declare Shadow DOM via HTML, without JavaScript. Declarative Shadow DOM is available in Google Chrome and Microsoft Edge. A simple polyfill brings Declarative Shadow DOM to browsers that have not implemented the specification.

The many uses of Web Components

Web Components have been growing in popularity since the main specification that drives the standard, Custom Elements v1, became available in all evergreen browsers in January 2020 with the release of the Chromium-based Microsoft Edge browser. In 2022, Google reported Web Component usage had almost quadrupled since 2019: 18% of websites include a custom element on the page[8].

Single page applications

Web Components don't replace all the functionality provided by a JavaScript framework. Frameworks provide abstraction around common web development tasks: routing, state management, data binding, and coding UI components. Web Components overlap only this last feature of JavaScript frameworks: the UI component.

You probably wouldn't develop a complex single page application solely with Web Components, opting to integrate custom elements with a JavaScript framework instead. If the desire is to code a single page application with minimal to zero external dependencies, Web Components require, at the very least, a routing mechanism that can instantiate and destroy custom elements. In Part Three of this book, we're going to cover how to use Web Components in the context of a single page application.

Standalone widgets

Sometimes you need to code a standalone user interface that shouldn't interact with the rest of the page and needs to work on several different sites. It doesn't make sense to use a JavaScript framework because you incur the cost of loading a dependency that may be larger in filesize than your implementation.

[8] https://chromestatus.com/metrics/feature/popularity#CustomElementRegistryDefine

Web Components work out extremely well in this scenario because you're coding with "vanilla" JavaScript. You can encapsulate the template of a custom element with Shadow DOM so your component can't easily interact with the surrounding page. If your task is to develop a standalone widget, you'll learn most of what you need in the first chunk of this book. Part One features several hands-on exercises where you'll explore the custom elements specification.

Static websites

Given the portability of custom elements, you can bundle custom elements and include them in a `<script>` tag. When the browser parses DOM, the custom element will be instantiated. If you want to avoid a flash of unstyled content, it's possible to server-side render custom elements. In Part Three, we're going to talk about server-side rendering Web Components using Declarative Shadow DOM.

So far I've talked about how Web Components behave and where you might find custom elements. In the next sections, I further define Web Components through the HTML specification, then talk about how custom elements can be viewed through the lens of a design system, followed by a guide for getting started developing with the repository that accompanies this book.

UI Library

The main appeal of Web Components is due to how custom elements are interoperable and portable. UI libraries built with Web Components make a lot of sense at large corporations that need to scale. If you want UI components to as compatible as possible with any framework, Web Components are well suited for that task. Don't take my word for it! Large corporations like Salesforce, Microsoft, and Google have adopted Web Components for their open source UI libraries.

Developing a UI library with Web Components is intuitive because the goals of the library align with the behavior of Web Components. UI libraries can gain greater adoption when they are portable, interoperable, and have a long lifespan. In Part Two of this book, we're going to focus on developing a UI library with Web Components.

Implementing a design system with Web Components

When developing user interfaces you'll most likely reference a mockup provided by a Designer. The UI component you're developing is part of a design system. The definition of "design system" is tricky, as different people have proposed vastly different definitions, but I'm going to try anyways.

A design system is a set of guidelines for the look and feel of a product or project. Design systems can feature a mission statement, design ethos, vocabulary, values and measurements often referred to as "tokens", components, layouts, and possibly more. The design system is a living document that defines reusable patterns engineers can reference when developing user interfaces.

The main touchpoints for a design system web developers should be concerned with are the following:

Tokens

Constants that define common reusable design patterns including color, size, spacing, typography, and more.

Components

Discrete units of user interface functionality that have a distinct purpose and behavior.

Layouts

Organization of components in a web site.

Conventions used in this Book

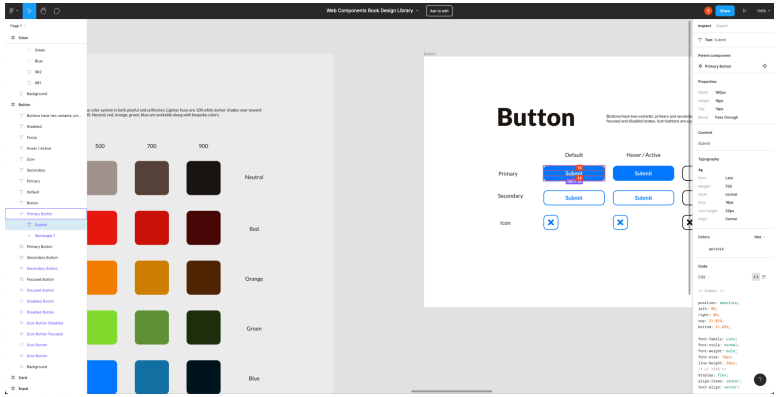
Each chapter of this book presents a problem statement and orientation, followed by acceptance criteria then a step-by-step walkthrough of the implementation, ending with a summary.

Figma

It's practical to look at Web Components through the lens of a design system. Web Components uses tokens as building blocks for developing a reuseable user interface. Web Components can display in different orientations or states: a layout. Web Components can also be part of another layout compromised of one or many components.

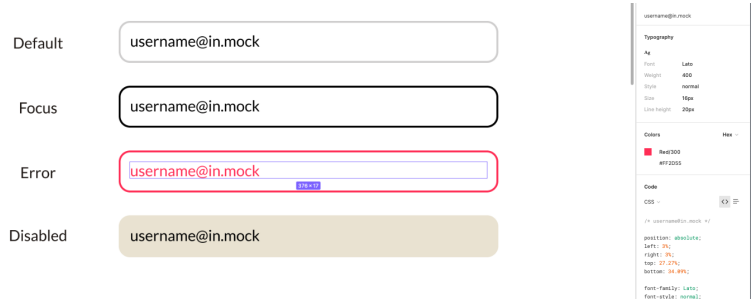
Design systems offer guidance for developing user interfaces. Throughout this book you'll reference the Fullstack Web Components Design System in Figma[9].

[9] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library



In Figma, you can reference different values of design tokens and inspect and measure parts of UI components and layouts. Take a moment and familiarize yourself with the Figma interface. On the left sidebar is a navigation menu that lists different layers in the focused artboard. When you click on elements in the artwork, various properties of the selected element are displayed in the right sidebar.

Whenever an element is selected, the color of that element is displayed in the right sidebar by token name followed by the hexidecimal value of the color.



A global CSS stylesheet provided in the development environment for this book contains CSS variables that map to each design token. To use the `Red/300` token, you'd reference the `--color-red-300` CSS

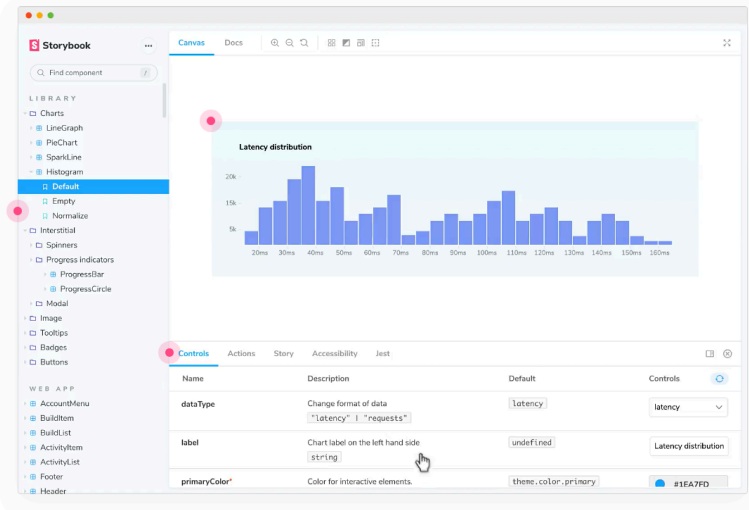
variable:

```
color: var(--color-red-300);
```

Similarly, CSS variables for sizing, border radius, font, icons and box shadow are provided on the global stylesheet found in src/style/style.css. You may reference these CSS variables when styling each UI component.

Storybook

The main development environment for a considerable portion of this book is **Storybook**[10], "an open source tool for building UI components and pages in isolation."



Storybook is run locally on your computer. In several chapters, you'll define Stories for the UI component you're coding. A Story

[10] https://storybook.js.org

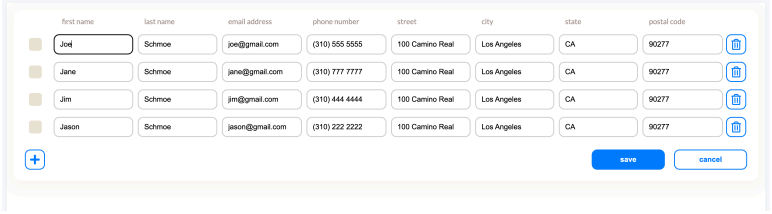
is a configurable template that hosts your component. Through the Storybook user interface, you can define different controls that change UI component behavior. By the end of this book you should have a good understanding of how to develop and maintain UI components in Storybook.

Development lifecycle of a UI component in Storybook

1. Reference the mockup in Figma
2. Define the story in Storybook
3. Style the component with design tokens
4. Develop behaviors with JavaScript / TypeScript
5. Write tests

Application

In the third part of this book, you transition from developing user interface components to developing with those components in the context of a web application. When developing the application, you'll change up the local environment to a custom Webpack server and only use Storybook when necessary.



The application you're going to develop is called "In", a web app used to store a user's contact list. After the user authenticates with a login form, a user interface displays a table filled with contacts.

The user can view or edit the contact list. User interfaces found in the application are developed in Storybook first, exported from a UI library package, then imported into the application source code.

Cypress

Cypress[11] is a test runner you'll use to write end-to-end tests for each UI component.

Storyshots

Storyshots[12] is a Storybook addon you'll use to generate automatic visual snapshots for each UI component.

Monorepo

When developing against the content in this book, you'll be working in a monorepo built with lerna[13]. Lerna allows you to manage multiple projects in the same repository. Those projects are defined as:

- `client` web application front end
- `common` reusable functions, utilities
- `component` UI library
- `server` web application back end
- `style` resources exported from the design system such as CSS styles, icons, fonts

[11] https://www.cypress.io
[12] https://storybook.js.org/addons/@storybook/addon-storyshots
[13] https://lerna.js.org

Throughout the book, you'll switch between different projects in the repository depending on what you're currently working on. When developing UI components, you'll work in the component directory. If you're developing the web application, you'll code in files found in the client or server directories.

Naming

- Class names are prefixed to avoid collisions with other UI component libraries.
- File names are in kebab-case.

Setting up the development environment

Getting started is easy. Great care was taken to setup a development environment that can be used throughout the book, allowing you to focus on the content of each chapter. To start development, clone the repository from Github, install project dependencies, and run a script that runs a local Webpack development server. Each of these steps are outlined below.

Prerequisites

The following tools should be installed prior to starting development.

Terminal application

Which Terminal you use depends on your system. MacOS and Linux systems come with the Terminal application. For Windows

10 and above, all Terminal commands featured in this book should be run with PowerShell and Windows Subsystem for Linux[14]. Follow these instructions to install Windows Subsystem for Linux[15]. Running node scripts in Windows Subsystem for Linux ensures the commands will work.

git

Changes to the codebase are tracked via git. git is a version control system that allows projects like this one to track changes across the codebase effectively. If you have little or no experience with git, don't worry. Each chapter clearly demonstrates the git commands you'll need to run to navigate between changes. git is installed differently depending on your system.

- Mac[16]
- Windows[17]
- Linux[18]

NodeJS

node[19] is a JavaScript runtime built from V8 JavaScript engine[20]. Whereas JavaScript is usually run in a web browser, node runs server-side. node is traditionally used for back-end services, but it's also used to code tooling for JavaScript projects. npm is a package manager for node. npm packages can be installed in any node project, allowing you to reuse code published on the npm registry in a local development environment.

[14] https://docs.microsoft.com/en-us/windows/wsl/about
[15] https://docs.microsoft.com/en-us/windows/wsl/install
[16] https://osxdaily.com/2014/02/12/install-command-line-tools-mac-os-x/
[17] https://gitforwindows.org/
[18] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[19] https://nodejs.org/
[20] https://v8.dev/

This book requires node 16 and above installed on your machine. It is highly recommended to install node and npm with nvm[21], the Node Version Manager. nvm allows you to install multiple versions of node locally, switching between them for each project on your system. Installation instructions for nvm can be found here[22].

JavaScript IDE

To actually code the various components and application in this book you'll need an IDE (integrated development environment). There are many free IDE, so don't feel like you need to spend a ton of money on one. A few popular IDE are listed below. Download and install your preferred application for JavaScript development.

- Visual Studio Code[23] (MacOS, Windows, Linux)
- Nova[24] (MacOS)
- WebStorm[25] (MacOS, Windows, Linux)
- Atom[26] (MacOS, Windows, Linux)

Once you have all of the above tools installed on your system, you're ready to clone the git repository, install project dependencies, and run the development server.

GitHub

The repository for this book is found on GitHub. If you don't already have a GitHub account, signup for a new account[27]. If you've never used GitHub before on your machine, you'll need to

[21] https://github.com/nvm-sh/nvm
[22] https://github.com/nvm-sh/nvm#installing-and-updating
[23] https://code.visualstudio.com/
[24] https://nova.app/
[25] https://www.jetbrains.com/webstorm/
[26] https://atom.io/
[27] https://github.com/signup

setup your username locally[28]. More information about setting up GitHub locally is found in the Set up Git Guide[29].

After you've properly setup git locally, fork the repository on Github[30], then clone your fork of the repository via the Terminal, making sure you run the command in a directory where code is usually stored on your system. If you don't already have any code projects on your computer, make a new directory to store this book's codebase, then run the following command, inserting your Github username in lieu of `{username}`.

```
git clone https://github.com/{username}/fullstack-web-com\
ponents.git
```

Change directory into the project root directory.

```
cd fullstack-web-components
```

There are times you may need to update your local repository from the upstream repository, mainly to uptake dependency upgrades or new content. Further instructions for maintaining your local fork of the repository are found in the README.md[31].

Yarn

This repository uses yarn[32] to install npm dependencies and run node scripts.

[28] https://docs.github.com/en/get-started/getting-started-with-git/setting-your-username-in-git
[29] https://docs.github.com/en/get-started/quickstart/set-up-git
[30] https://github.com/readymade-ui/fullstack-web-components
[31] https://github.com/readymade-ui/fullstack-web-components
[32] https://yarnpkg.com/

```
npm install --global yarn
```

Once yarn is installed you can install project dependencies with the `yarn` command.

```
yarn
```

Storybook

After project dependencies are installed with yarn, run the `storybook` script to start up the local Storybook development environment.

```
yarn storybook
```

If Storybook appears in the default browser on your computer, you have correctly setup the development environment for this book.

Summary

Through the practical exercises in this book, you'll learn how to code user interfaces that are interoperable, reusable, accessible, portable and require zero dependencies. The goal is to mirror best practices at enterprise corporations that you can bring into your own professional development.

Part One Specification

In the first part of this book, you're going to take a deep dive into the Web Components specification by developing the UI components that comprise a login form. Through developing with the different types of custom elements and several other parts of the Web Components specification, you'll learn:

- How to encapsulate template and styling with Shadow DOM
- How to provide dynamic templates for a UI component
- The lifecycle of a custom element
- Event handling and delegation
- How to make custom elements accessible
- How to architect complex form validations

Part Two UI library

In the second part of this book, we focus on developing Web Components in a UI library.

- We'll use TypeScript to bring a Decorator pattern to enhance component development, simplifying common tasks with a standardized, opinionated interface.
- Event handling is taken to a whole new level when we develop a table component with custom elements.
- We'll focus on the benefit of reusable HTML templates when developing a panel component.
- Later, we'll implement a theme across the UI library using CSS Shadow Parts.
- Finally, we'll optimize the UI library for distribution by bundling every component with Rollup.

Part Three Application

In the third part of the book, we take a look at how Web Components are used in the context of a web application. Lerna is introduced to the development environment to coordinate development of several packages in a monorepo.

- First we'll code an application shell with custom elements
- Then we'll handle routing different views in our application

- We will follow that with a lesson about server-side rendering custom elements with Declarative Shadow DOM.

In the final chapter of this book, we'll summarize what was learned and look to the future of Web Component development.

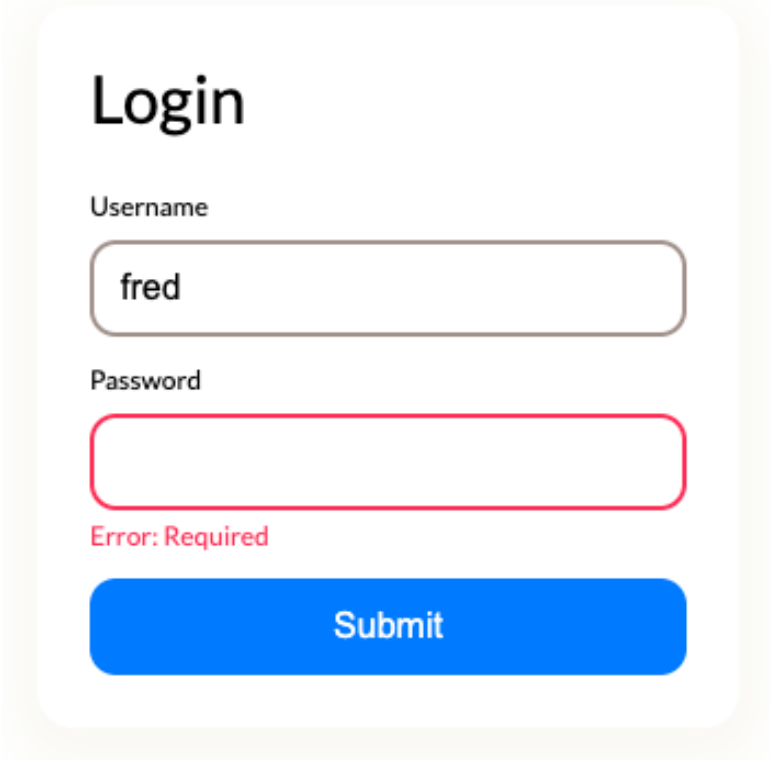
In the next chapter **Getting To Know Custom Elements**, you will learn the complete development lifecycle of an autonomous custom element.

Part One

Part One

Specification

In Part One of this book we'll explore several Web Components specifications by developing the UI components that make up a login form. This login user interface will be used in Part Three so users can authenticate with the application.



What you will learn

While developing the login use interface with Web Components, you'll explore the following concepts:

- How to encapsulate template and styling with Shadow DOM
- How to provide dynamic templates for a UI component
- The lifecycle of a custom element
- Event handling and delegation
- How to make custom elements accessible

- How to architect complex form validations
- The differences between autonomous, customized built-in, and form-associated custom elements

Developing in Part One

You'll use Storybook as the primary development environment throughout Part One. For each chapter, run `yarn storybook` in the Terminal. Every change made to the codebase will be reflected in the Storybook UI while Storybook is running in Terminal. When you are finished developing, cancel the server by holding down Ctrl on the keyboard and pressing C.

Getting to Know Web Components

Chapter One

Introduction

You can make completely customized HTML elements with autonomous custom elements, so it can be daunting to know where to start. In this exercise, autonomous custom elements are highlighted for what they do best: provide a means to bind template encapsulation provided by Shadow DOM with a unique HTML tag and then reuse the custom element throughout DOM.

Autonomous custom elements are the most ubiquitous of the three types of custom elements due to their compatibility in all major evergreen browsers including Apple Safari, Google Chrome, Microsoft Edge, and Mozilla Firefox. When building out a UI library, you'll possibly find yourself coding with autonomous custom elements the most because of their utility.

In addition to autonomous custom elements and Shadow DOM, we'll make use of HTML templates and CSS variables in this chapter.

What you will build

In this chapter, you'll code a reusable container called the Card component. This component can contain a lot of different templates, from user profile metadata to a sign-up form, so the template

has to be flexible enough to accept virtually any content, but rigid enough so that layouts adhere to design guidelines found in the design system.



Food

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Read

What you will learn

- What is an autonomous custom element?
- What is Shadow DOM and why should I use it?
- What is HTMLTemplate and why should I use it?
- What are template slots?
- What are CSS variables?
- How do I use design tokens to style a custom element?
- How do I style slotted components?
- How do I include a dynamic template in a custom element?
- How do I configure controls for a story in Storybook?
- How do I define a story in Storybook?
- How do I write integration tests for autonomous custom elements with Cypress?
- How do I make snapshots of autonomous custom elements with Storyshots?

Acceptance criteria

- A class called `CardComponent` is exported from packages/component/src/card/Card.ts
- CardComponent can accept dynamic images, paragraphs, links, and headlines in the template
- CardComponent is styled per the mockup in Figma
- CardComponent is configurable via Storybook Controls
- Integration tests are found in cypress/integration/-Card.spec.js
- Snapshot tests validate the design implementation of `CardComponent`

Getting started

Navigate to the project directory. The default name of the directory is fullstack-web-components from when you cloned the git repository.

```
cd fullstack-web-components
```

Checkout a new branch named `chapter-1`. You'll want to keep the master branch clean in case any updates trickle in from the main repository.

```
git checkout -b chapter-1
```

Run Storybook:

```
yarn storybook
```

Scaffolding CardComponent

In this first section, you'll code an autonomous custom element from scratch. You'll use Storybook as a development environment. If you haven't already, navigate to the project directory and run `yarn storybook`. Have no fear if you've never coded a custom element before because this book walks you through each exercise step by step.

Generate the card component

In the root directory of the project, make a new directory at `src/component/card`. `mkdir` is a good command line utility for

this. To run other commands in your Terminal while Storybook is running, you'll need to open another Terminal window or tab and navigate to the project directory.

```
mkdir -p packages/component/src/card
```

In the card directory, make a new file and name it `Card.ts`.

```
touch packages/component/src/card/Card.ts
```

Here we're borrowing from a convention of the React style guide, preferring to name the component files with an uppercase first letter. This is how files throughout this book will be named; however, you may name components using whatever convention you desire.

If you haven't already, now is a good time to open the new `Card.js` file in your favorite IDE.

In `Card.js` export a `class` called `CardComponent` that extends from `HTMLElement`. We need to export this class so we can use it in Storybook later. All autonomous custom elements extend from the same base class: `HTMLElement`, and must include a constructor that calls `super()` first, in order to establish the proper prototypical chain.

```
export class CardComponent extends HTMLElement {
  constructor() {
    super();
  }
}
```

Following the call to `super()`, define the template for the Card. Instantiate a Shadow DOM template for this component by creating a `const` called `shadowRoot` and making it equal to `this.attachShadow({ mode: 'open' })`.

```
    const shadowRoot = this.attachShadow({ mode: 'open' }\
);
```

This establishes a Shadow Root for this component, which encapsulates any DOM nodes inside the ShadowDOM tree from the rest of the DOM. We can call `attachShadow` because it exists as a method on the `HTMLElement` class, available because we extended from `HTMLElement`. `attachShadow` is also available on any HTML element that acts as a container like `<article>`, `<div>`, `<header>`, `<h1>`, and even `<p>`. We're going to define our own HTML tag though later in this section.

`attachShadow` takes an argument which is an Object that configures how ShadowDOM behaves. By setting the property `mode` to `'open'`, we're allowing access to the Shadow Root via JavaScript and because of this setting when we reference `element.shadowRoot`, it will return an `Object`. If `mode` is set to `'closed'`, `element.shadowRoot` will return `null`. We'll explore use cases for closing the Shadow DOM later, but for now it's safe to keep the ShadowDOM tree open.

Now that we've established this custom element should use Shadow DOM, let's give it a template using `HTMLTemplateElement`. In the `constructor`, following the `shadowRoot`, add the following lines of code:

```
    const template = document.createElement('template');
    template.innerHTML = `
      <span>Hello World</span>
    `;
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
```

HTML templates are reusable DOM trees that are not parsed until they are instantiated. In this case, the HTML template will be instantiated when an instance of the Card component is rendered and a clone of the template node is appended tp the Shadow Root

`appendChild` exists on the Shadow Root, which behaves similarly to any other DOM node.

To verify the component is working initially, include a simple "Hello World" in the template. The `CardComponent` class should look like this when you are finished.

```
export class CardComponent extends HTMLElement {
  constructor() {
    super();
    const shadowRoot = this.attachShadow({ mode: 'open' }\
);
    const template = document.createElement('template');
    template.innerHTML = `
      <span>Hello World</span>
    `;
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
  }
}
```

Finally, establish the tag name for the custom element by registering the class you just defined with the CustomElementRegistry.

```
customElements.define('in-card', CardComponent);
```

This last step is significant because without it, the browser can't interpret `CardComponent` as the `class` associated with the HTML tag `<in-card>`. We prefixed the tag name with `in-` because that is the name of the application we're coding: In. The tag name for custom elements is limited to kebab case and can contain at least one dash. Prefixing the component differentiates it from components from other design systems that may exist on the same website, so naming the components is something we want to get right fairly early on.

Define the story

To visualize the Card component in Storybook, we need to define a story in a new file.

In the `card` directory, make a new file and name it `Card.stories.js`.

```
touch packages/component/src/card/Card.stories.js
```

In this file, import the `CardComponent`.

```
import { CardComponent } from "./Card";
```

Next we should configure where the stories defined here will be displayed in the Storybook UI.

```
export default {
  title: "Components/Card",
};
```

Storybook expects any file matching `*.stories.js` to export a configuration `Object` known as Component Story Format[33] as the `default export`. At the very least, this configuration includes a `title` property with the name of the directory as it should be displayed in the Storybook sidebar. Categorize stories further with slashes. The above example configures Storybook to display a collapsible directory called Components, with a nested directory called Card in the Storybook sidebar. At least, Storybook will display this directory once we've defined some stories.

Define a template for Storybook. Templates in Storybook are functions that return a template. In our case, we want to define a `const` called `PrimaryTemplate` and return `<in-card></in-card>` from the function. Ensure the element is properly formatted with start and end tags as this is a requirement for autonomous custom elements.

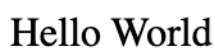
[33] https://storybook.js.org/docs/web-components/api/csf

```
const PrimaryTemplate = () => `<in-card></in-card>`;
```

Export the template, naming the Story `ImageCard`, then call the `PrimaryTemplate` function with `bind`, passing an empty `Object` for now. Later, we'll add arguments to the `PrimaryTemplate` function that will be passed to the story using `bind`.

```
export const ImageCard = PrimaryTemplate.bind({});
```

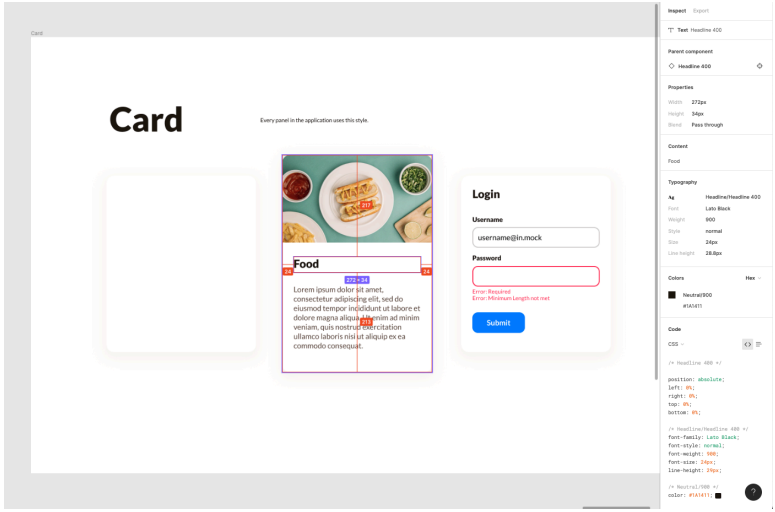
In a browser, navigate to http://localhost:6006/, open the "Components" directory in the sidebar navigation, find and click on "Image Card", and you should see "Hello World" in Storybook.



For the rest of this chapter, you'll work in both `Card.js` and `Card.stories.js`. The implementation for the `CardComponent` is defined in `Card.js`, while the `Card.stories.js` file contains stories that demonstrate various behaviors of the component.

Structure content with template slots

Designers have delivered a mockup of the Card in Figma, so let's take a look[34].



The mockup reveals at least two different layouts that can be embedded in the component's template. Patterns emerge as we review the mockups. There is a stylized headline that can be followed by a paragraph, or some other content like a form, an image can display full width or appear as an avatar, centered horizontally in the Card layout.

As we discuss the layout with the designer, we settle on a reusable layout for the Card component that could fulfill all of the proposed requirements of the design. We're going to implement a "header-content-footer" layout for the Card.

Luckily, Web Components include an API that allows for dynamic

[34] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=331%3A24

content called slots. Slot elements make templates flexible, allowing engineers to inject different content in the same slot.

If there is a slot declared in a Shadow Root like this with the `name` attribute:

```
1          <slot name="header"></slot>
```

anyone using the template in the context of Shadow DOM can inject other elements which include the `slot` attribute that specifies a slot with a matching `name` attribute.

```
1    <h4 slot="header">${headline}</h4>
```

When employing HTML template slots in the context of Shadow DOM, some interesting side-effects happen. Slotted content is accessible through Light DOM; anyone could `document.querySelector('slot="header"')` and find an element that matches the `<h4>` element in the above example. This side-effect is not guaranteed, as deeply nested slots, those encapsulated by multiple Shadow DOM boundaries, won't bubble their content up to the Light DOM. Any elements declared in the HTML template encapsulated by

Shadow DOM are typically inaccessible to Light DOM, although since the mode is set to `open` in this particular component, any element in Shadow DOM could be queried from Light DOM on the `shadowRoot` property of the `host:` which in our case would be the instance of `<in-card>`.

A header-content-footer layout is achieved with HTML elements using semantic HTML: `<header>`, `<section>` and `<footer>`, respectively. These elements will be scoped to the Shadow DOM of the Card component. By declaring slot elements inside each element, users of the Shadow DOM can project customized content into the header, content, and footer. Since we named the slots intuitively,

engineers should understand where the content they are injecting into the template will appear.

In `Card.js`, remove the 'Hello World' content and include a new template that uses HTML template slots.

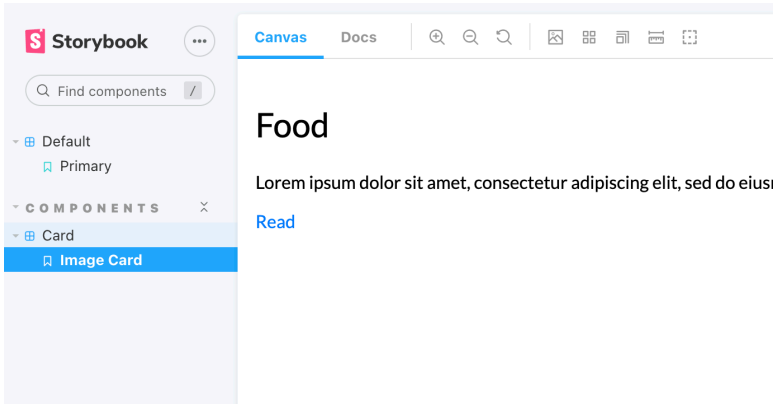
```
      <header>
        <slot name="header"></slot>
      </header>
      <section>
        <slot name="content"></slot>
      </section>
      <footer>
        <slot name="footer"></slot>
      </footer>
```

If you notice the 'Hello World' disappears in Storybook, that's expected because we still have to populate the new template slots with some content.

In `Card.stories.js`, update the default `ImageCard` story with a new template that injects content into each of the slots. You can source text from Figma by clicking on the element in Figma and clicking 'Copy' in the Content area of the right sidebar. For each of the elements, declare the appropriate tag and include the `slot` attribute, setting the value to the template slot the element should appear in.

```
const PrimaryTemplate = () => `
<in-card>
  <h4 slot="header">Food</h4>
  <p slot="content">Lorem ipsum dolor sit amet, consectet\
ur adipiscing elit, sed do eiusmod tempor incididunt ut l\
abore et dolore magna aliqua. Ut enim ad minim veniam, qu\
is nostrud exercitation ullamco laboris nisi ut aliquip e\
x ea commodo consequat.</p>
  <a href="#" slot="footer">Read</a>
</in-card>`;
```

Notice the changes in Storybook.



Looking back at the mockup, it appears the `CardComponent` header should support images, may be a single image displayed full width. The styling of this image will be handled in the next section of this chapter. For now, we need to figure out how to inject an image using HTML template slots.

HTML template slots can accept more than one element, rendering each element in the order they are received. If the image should be displayed above the headline, inject the `<img>` before the `<h4>`, but ensuring they both include an attribute that links each element to the header slot: `slot="header"`. Set the `src` attribute to the

appropriate image sourced from Unsplash.

```
<img
  slot="header"
  src="https://images.unsplash.com/photo-1612392167062-8f\
76710986ba?ixid=MnwxMjA3fDF8MHxwaG90by1wYWdlfHx8fGVufDB8f\
Hx8&ixlib=rb-1.2.1&auto=format&fit=crop&w=1350&q=80"
/>
<h4 slot="header">Food</h4>
```

The image should appear large because we haven't styled the component yet. The image is merely following the default flow of the document.

That's all there is to the structure of the `CardComponent` template for now. Later in this chapter, we'll add a `<style>` tag to the template when we style the component. First though, we're going to wire up Storybook to view several variants of the content projected into `CardComponent`.

Create controls for the story

The content of the Card is projected through slots, however; we've declared the content as static HTML in the Image Card story. Storybook Controls allow users to change the content from the Storybook UI. With Controls, you can dynamically change the headline, content appearing in the section, the text that appears as a link, or even the image that is displayed. Storybook Controls enable anyone to test how components behave.

Controls are configured in the same file as the story: `Card.stories.js`. The values set by each Control are passed through arguments to the template function, in our case `PrimaryTemplate`. Default values for controls can be configured for each story.

Configure the Card story with controls that map to the elements that are projected into each slot. Add an `argTypes` property to the configuration and make it equal to an empty `Object`. The properties on `argTypes` map to what is displayed as control names in Storybook, by default. Each control is configured through another `Object`, where the type of control is declared. Each control should have a `type` property set to `'text'`. This will make Storybook display text fields in the Controls section of the user interface for each of the controls declared here. The user will be able to enter in a string for the image url, headline, content, and label for the hyperlink.

```
export default {
  title: 'Components/Card',
  argTypes: {
    image: {
      control: { type: 'text' },
    },
    headline: {
      control: { type: 'text' },
    },
    content: {
      control: { type: 'text' },
    },
    link: {
      control: { type: 'text' },
    },
  }
};
```

When properly configured, Storybook will display controls with empty text fields in the area below the card. To set default values for each control, set the `args` property on `ImageCard` to an `Object` that contains properties named for each control that set the value of each control.

```
ImageCard.args = {
  image: 'https://images.unsplash.com/photo-1612392167062\
-8f76710986ba?ixid=MnwxMjA3fDF8MHxwaG90by1wYWdlfHx8fGVufD\
B8fHx8&ixlib=rb-1.2.1&auto=format&fit=crop&w=1350&q=80',
  headline: 'Food',
  content: 'Lorem ipsum dolor sit amet, consectetur adipi\
scing elit, sed do eiusmod tempor incididunt ut labore et\
 dolore magna aliqua. Ut enim ad minim veniam, quis nostr\
ud exercitation ullamco laboris nisi ut aliquip ex ea com\
modo consequat.',
  link: 'Read'
};
```

For `PrimaryTemplate` to reference the controls, first add each control as an argument of the `PrimaryTemplate` function.

```
const PrimaryTemplate = ({ image, headline, content, link\
 }) => `
```

Since `PrimaryTemplate` returns a template string, use template variables to set the value of each control.

```
const PrimaryTemplate = ({ image, headline, content, link\
 }) => `
<in-card>
  <img slot="header" src="${image}">
  <h4 slot="header">${headline}</h4>
  <p slot="content">${content}</p>
  <a href="#" slot="footer">${link}</a>
</in-card>`;
```

Test out setting each control's value until you are satisfied everything is wired up correctly. If you don't see the Controls in Storybook, click the **Shortcuts** button next to the Storybook logo in the top left of the browser window and ensure **Show addons** is

enabled or press 'A' on your keyboard. Another trick if the Controls still don't appear is to click **Change addons orientation** in the same menu or 'D' on your keyboard. Change the value of any of the text fields to observe the Card component template change in Storybook.

Commit your changes in git to complete this section. For those who are newer to git, this command may appear like below in Terminal.

```
git commit -a -m "feat: card scaffold"
```

Now that the template for the component is established and everything is wired up in Storybook, it's a good time to start styling the component.

Styling

Shadow DOM encapsulates the component's template, presenting an interesting challenge when styling autonomous custom elements. CSS declared in Light DOM, like in a `<style>` element in the `<head>`, cannot pierce the Shadow DOM boundary. Likewise, if we include a `<style>` element with CSS declarations in the component template, those styles are not accessible in Light DOM. CSS declared in the context of an autonomous custom element that uses Shadow DOM is encapsulated, the styles cannot leak outside of the component.

The ::slotted pseudo selector

We've chosen to project content through the Shadow DOM boundary using HTML template slots, which possess a whole other set of challenges. Elements projected through slots belong to the Light DOM and can't be styled directly in the component template unless we use a special CSS pseudo selector for slotted content: `::slotted`.

Deciding whether to style slotted content in the context of Shadow or Light DOM can be confusing. Elements in Light DOM can be styled with a global stylesheet. That's fine for styles that are heavily used throughout an application, however this technique falls apart when a custom element is nested under multiple Shadow DOM boundaries. We won't run into this limitation in this exercise because we're working with only one layer of Shadow DOM. The `::slotted` selector comes in handy for styling the layout that is contextual to how the element displays in relation to the component.

An example of this is how the `<a>` element is styled. In the global stylesheet, all links get their "look" when selecting `a:link` in CSS. Since the `<a>` element is projected through the Shadow DOM, it picks up the styling of the closest global stylesheet. The following example found in `packages/style/style.css` styles all links with a `font-family`, `font-weight`, and `font-size` using CSS variables to set the values in the style declaration. This stylesheet is loaded in the `<head>` of the Storybook iframe where components are viewed. Storybook is configured to load this global stylesheet in the same `<iframe>` as the story. That's why you observed font styling on the content projected through the template slots in the last section.

```
a:link, a:visited {
  font-family: var(--font-default);
  font-weight: var(--font-weight-default);
  font-size: var( --font-body-md);
}
```

`<a>` is styled with `display` set to `inline` by default, but in the case of this component, we'll want to ensure the link behaves more like a `block` element. This is because the link should display properly in a vertical layout, beneath the paragraph. We can override the style of `<a>` just for this component using the `:slotted` pseudo selector.

```
::slotted(a:link), ::slotted(a:visited) {
  display: block;
}
```

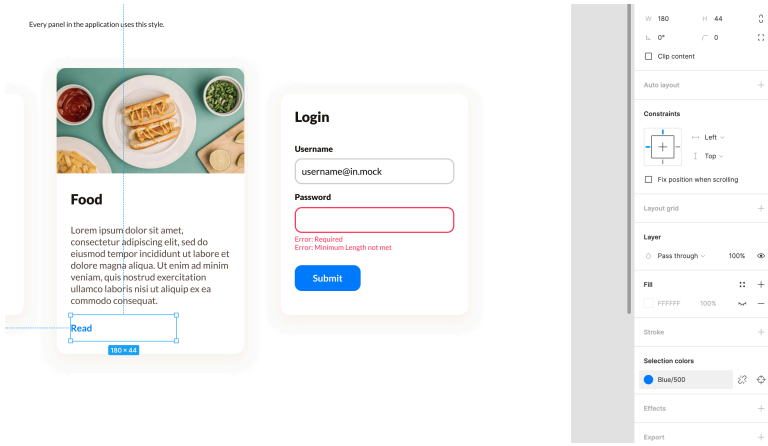
Styling slotted elements with the `:slotted` pseudo selector is useful for maintaining a component's styles so they are completely self-contained. The `<a>`, `<p>`, and `<h4>` included in the current template are exposed to Light DOM for now, but what if the component was encapsulated by another component's Shadow DOM? These elements would lose their styling, unless the same styles were defined in the parent component's stylesheet. The parent element's Shadow DOM blocks styles in the global stylesheet. We'll explore this scenario in later chapters, but for now the component can find styles in a global stylesheet just fine for `<a>`, `<p>`, and `<h4>`.

CSS variables

Global style declarations can't be applied to elements found in the autonomous custom element Shadow DOM tree; however, any CSS variable declared in a global stylesheet can pierce the Shadow DOM boundary. CSS variables, otherwise known as CSS custom properties, allow you to reuse the value of a CSS property, while maintaining that value in one place. CSS variables are declared with custom property notation, prefixed with `--` and later accessed with the `var` function. The name of each CSS variable in `style.css` is derived from its function in the design system: `--color`, `--font`, `--margin` to name a few. The CSS variables declared in `style.css` map directly to design tokens found in Figma. Stylesheets like this one could be generated by a design engineer so engineers can reference the same variables in code observed in Figma mockups.

```
  --color-black: rgba(0, 0, 0, 1);
  --font-default: 'Lato', sans-serif;
  --font-headline: 'Lato Black', sans-serif;
  --font-weight-default: 400;
```

In Figma, observe how each CSS declaration in the Inspector includes static values or a token. In the lerna monorepo, a global stylesheet is found in the packages/style directory. These tokens are exported from the global stylesheet as CSS variables so they can be reused in any component. The stylesheet is found in the file packages/style/style.css.



To begin styling the component, create a `<style>` element at the beginning of the template in Card.js.

```
  template.innerHTML = `
    <style></style>
    <header>
      <slot name="header"></slot>
    </header>
```

Any style declarations included in this `<style>` element will be

scoped to the component's Shadow DOM, encapsulated by the shadow boundary of the `CardComponent`.

Styling with :host

The first style declaration for the component will affect the outer most element's look. According to the mockup in Figma, the card's container should have a background color, rounded corners, and drop shadow, and have a maximum width of `320px`. To style the container (the `<in-card>` element), use a pseudo selector called `:host`. The `:host` pseudo selector is specific to Shadow DOM and it selects the shadow host of the Shadow DOM we're currently working with.

Style the component using the `:host` pseudo selector, using CSS variables found in `style.css` that map to design tokens or measurements observed in Figma. For instances in Figma when you can't find a token and all you have is a measurement, find the corresponding token in `style.css` by searching for its value in your IDE.

```
<style>
:host {
  display: block;
  background: var(--color-white);
  border-radius: var(--radius-md);
  box-shadow: var(--shadow);
  overflow: hidden;
  max-width: 320px;
}
```

Immediately, you can observe some results in Storybook. The height of the card is still out of sorts, but constraining the width and setting a `border-radius` has made a noticeable improvement. Content has also picked up the appropriate `font-family`.

Styling content

When looking at Figma, it appears the content inside of the card has padding on the left and right. At first, it seems prudent to style the `<header>`, `<section>` and `<footer>` with this padding, but there's the case of full width images that have to be supported. If the padding is on these container elements, the `<img>` element could not display full width. In later chapters, elements in Shadow DOM will be styled using typical CSS selectors, but for this implementation we're going to stick with the `::slotted` pseudo selector for remaining styling because this component allows for dynamic content with slots.

Styling slotted content

To give every slotted element some padding on the left and right, use the * selector, meaning 'all'.

```
::slotted(*) {
  padding-left: var(--padding-lg);
  padding-right: var(--padding-lg);
}
```

Next, override the default browser styles for a link, by selecting every `a:link` and `a:visited` and setting the `display` to `block`.

```
::slotted(a:link), ::slotted(a:visited) {
  display: block;
}
```

There seems to be a margin below the last element in the card. Target the last slotted element with the `:last-child` pseudo selector and apply the margin specified in Figma using a CSS variable.

```
::slotted(:last-child) {
  padding-bottom: var(--margin-lg);
}
```

To support full width images, override the `padding-left` and `padding-right` by selecting `img` and declaring a new `padding-left` and `padding-right`, making sure the `width` is set to `100%`.

```
::slotted(img) {
  width: 100%;
  padding-left: 0px;
  padding-right: 0px;
}
```

When you are finished styling, the Card should appear like this. Commit your changes in git to complete this section.



Food

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Read

The implementation of the Card is nearly complete. In this section we styled the `CardComponent` using CSS variables that pierce the

shadow boundary of the element, targeting the element with the `:host` selector while selecting elements slotted into the HTML template with the `:slotted` pseudo selector. You'll have more practice styling custom elements with the same techniques in following chapters. We've established the structure and presentation of this autonomous custom element, but before we close this chapter, let's add some automated tests.

Testing the card

The importance of testing UI components cannot be understated. You spent some time implementing a stable template for `CardComponent` and want to ensure it remains intact. That template has a API: the names of the slots engineers expect to project content into: header, content, and footer. Suppose you need to implement a new feature, inadvertently altering the original contract engineers relied on while using the `CardComponent`. Integration tests can warn you of breaking changes to the component.

While integration tests verify how the different parts of the code work together, snapshot tests alert you to differences in the look and feel of a component after changes to the codebase. Snapshot tests take an impression of the visual appearance of the component at a point in time and then check for differences over time. Throughout this book, you'll be creating and updating visual regression tests.

If you're new to testing, the role of tests will become more apparent in later chapters after you refactor the components in Chapter Four, add features or alter user interface behaviors, but for now, let's use integration tests to verify the `CardComponent` implementation. In this section, you'll write integration tests that check if `CardComponent` has the proper HTML template slots declared in the component's template. You'll verify that the content set by the default Storybook Controls is present. After writing these

integration tests with Cypress, you will generate a snapshot of the `ImageCard` story using the Storybook StoryShots plugin.

Integration tests

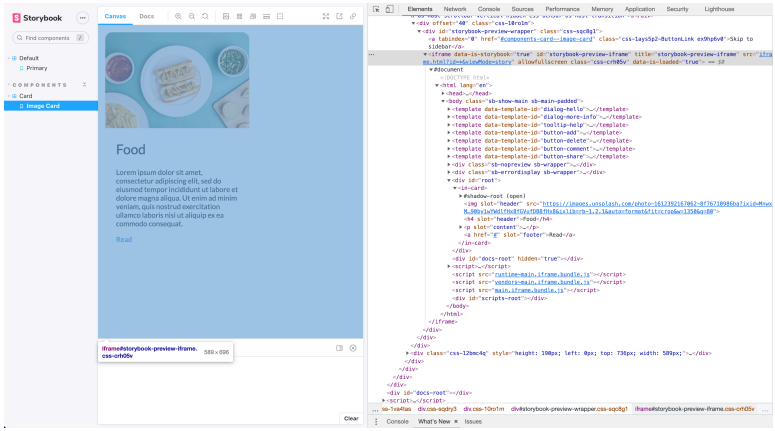
Cypress[35] is available in the repository for writing automation tests. You're not expected to know Cypress prior to reading this book, so we're going to walk through the process of writing tests with Cypress. The actual test files are located in `cypress/integration`. In the beginning, this directory is empty because no tests have been written yet.

Cypress provides robust framework and tooling for developing integration tests; however, Storybook is still the test bed. Cypress will navigate to views in Storybook and test various parts of each component. If you don't already have Storybook running, start it now in a Terminal window.

```
1  yarn storybook
```

Just like when you click on the Storybook navigation in the sidebar to navigate to the Image Card story, Cypress can navigate to a similar view to test the `CardComponent` in the browser. Storybook stories are displayed in an embedded `<iframe>` in the Canvas tab of the Storybook UI. If you inspect the Story using Dev Tools in Google Chrome, you'll notice the `<in-card>` tag wrapped by a `<body>` tag inside a `<iframe>`.

[35] https://www.cypress.io



Storybook provides a way to view stories without the rest of the Storybook UI. By allowing tools like Cypress to navigate to the story in isolation, the test can be more reliable because the test framework may not mistakenly query for elements in the Storybook UI. To see this in action, observe the path in the address bar when you navigate to the Image Card story.

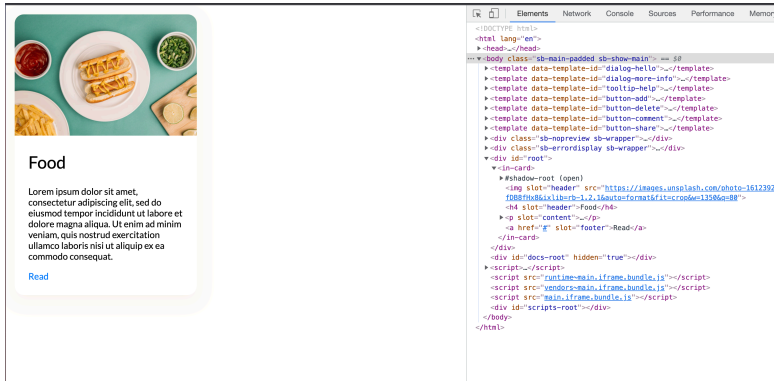
```
http://localhost:6006/?path=/story/components-card--image\
-card
```

The part after `/story/` corresponds to the title of the story you defined earlier (`Components/Card`), followed by the name of the story (`ImageCard`). Both are converted from pascal case to kebab case to satisfy the requirements of setting the `path` query param in the address bar.

If you reconfigure the query param to instead query for `id` instead of `path`, remove the `/story/` and insert `iframe.html` to the path after the hostname and port and before the query params, you'd end up with something like the below url in the address bar.

```
http://localhost:6006/iframe.html?id=components-card--ima\
ge-card
```

This address navigates the browser to a view that displays the Image Card story without the rest of the Storybook UI. This is ideal for test frameworks so the component implementation is tested in isolation.



You should understand this because Cypress is configured to test user interfaces running at http://localhost:6006, the same address as Storybook. When coding Cypress tests, you'll direct the test framework to navigate to the story before any test assertions. You only need the path after the host name and port to specify the address Cypress should visit in the browser prior to running any integration test.

```
    cy.visit("iframe.html?id=components-card--image-card"\
);
```

For a description of writing, running, and debugging Cypress tests, reference the online documentation[36]. To write the integration tests for `CardComponent`, follow the instructions below.

In a separate Terminal window from Storybook, create a new test file in the `cypress/integration` directory named `Card.spec.js`.

[36] https://docs.cypress.io/#Writing-tests

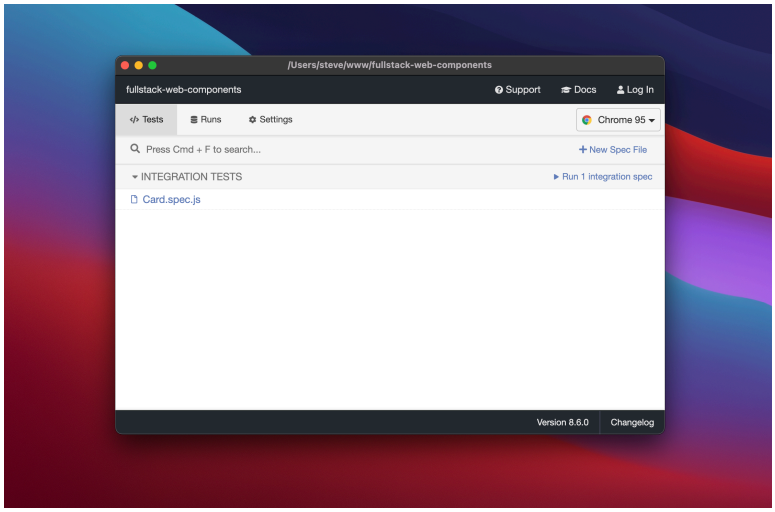
```
touch cypress/integration/Card.spec.js
```

In the same Terminal window, run Cypress in "open mode". A built-in script called `test:e2e:open` is available that calls `cypress open`.

```
yarn test:e2e:open
```

If this is your first time running Cypress, you may be prompted by the operating system to install or verify Cypress. Follow any onscreen prompts until you can run `yarn test:e2e:open` smoothly. Cypress may not be installed globally on your system and that's fine because it should be installed locally to the project. If you have difficulty running `cypress install`, try running the same command locally: `node_modules/.bin/cypress install`.

When Cypress runs in "open mode" the test framework opens a new browser window that includes a user interface for running integration tests.



If you try to run the `Card.spec.js` integration test right now by clicking on the link in the browser window, nothing would happen because we haven't written any tests yet. Lets remedy that.

All Cypress tests are wrapped in a `describe` block like below. Go ahead and write this `function` in `Card.spec.js`, passing in the name of the component as the first argument and a `function` callback in the second argument. The callback is where you'll write the tests.

```
describe("CardComponent", () => {});
```

Inside the `function` block, call the `it` method with the following requirements. This `it` function is available via Cypress. Pass in the name of the test as the first argument, followed by another `function` callback in the second argument. It's a good idea to provide a name here that describes what this particular test does. For the first test, you'll check there are `slot` elements in the `CardComponent`, so name the test `should find slots in the card`.

```
describe("CardComponent", () => {
  it("should find slots in the card", () => {});
});
```

Inside the `it function` callback, using the `cy` keyword navigate Cypress with the `visit` method, calling this method with the url you figured out earlier that navigates to the Image Card story.

```
    cy.visit("iframe.html?id=components-card--image-card"\
);
```

Cypress includes methods for querying DOM nodes, accessing Shadow DOM, and running assertions against DOM nodes on the same `cy Object`. Some methods you'll use to assert the slots are located in the Shadow DOM template are as follows:

Method	Purpose
`get`	Queries for a DOM element using a selector similar to `document.querySelector`
`shadow`	Accesses an element's Shadow Root
`find`	Queries for a DOM element inside the context of the query ran with `get`
`should`	The assertion for the current test

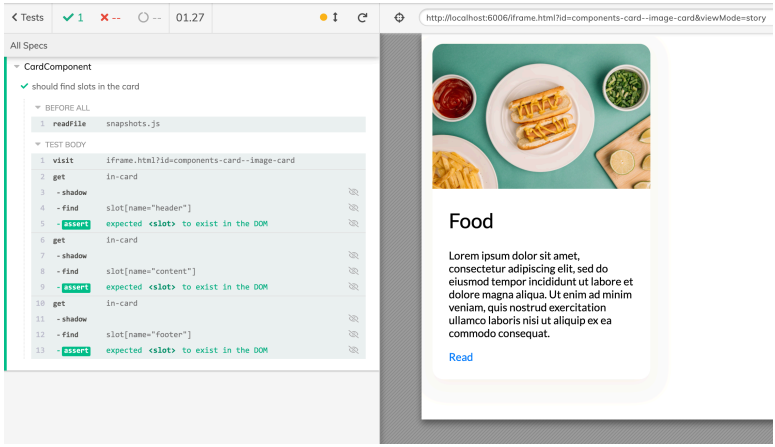
To test that the Shadow Root of the `CardComponent` instance includes the slots we intended, use the `get` method to query for the tag name `in-card`, then access the Shadow Root with a call to `shadow()`, chaining another call to `find`, passing in the `slot[name="header"]` as the selector. Finally call `should` with the string `exist`. This statement basically says find the `in-card` element, access its Shadow Root, then find the `slot` with the attribute `name` set to `header`, and check it exists. Repeat the same process for all three of the slots declared in the Shadow Root.

```
    cy.get("in-card").shadow().find('slot[name="header"]'\
).should("exist");
    cy.get("in-card").shadow().find('slot[name="content"]\
').should("exist");
    cy.get("in-card").shadow().find('slot[name="footer"]'\
).should("exist");
```

The test should look like the below code snippet when you are done. Run the test in the browser window by clicking the `Card.spec.js` link to verify the test passes.

```
  it("should find slots in the card", () => {
    cy.visit("iframe.html?id=components-card--image-card"\
);
    cy.get("in-card").shadow().find('slot[name="header"]'\
).should("exist");
    cy.get("in-card").shadow().find('slot[name="content"]\
').should("exist");
    cy.get("in-card").shadow().find('slot[name="footer"]'\
).should("exist");
  });
```

This test ensures if an engineer changes the names of the slots, the test will fail. A breaking change like that to the component could upset everywhere that instantiates an instance of the `CardComponent` in the application. That is well and good for testing the slots we've defined exist, but what about testing if the content gets correctly projected inside the slot? This is an integration test after all.



Before, you had to traverse the shadow boundary to test if the slot existed, but the actual content going into the slot exists in Light DOM, not Shadow DOM. You'll need a different way to query for the content when testing if the image exists and text content

contains the correct characters. That means you'll need to query for an element that is a parent node in DOM to `<in-card>`. Stories are contained by an element with the id `#root` in Storybook. Use that element to `find` the elements that are projected into each slot by selecting each element by the `slot` attribute.

Declare another test using the `it function`, this time specifying the test `should find the card slot content in light DOM`.

```
  it("should find the card slot content in light DOM", ()\
 => {
```

Call `cy.visit` with the same address as the first test.

```
    cy.visit("iframe.html?id=components-card--image-card"\
);
```

The first test assertion checks an `img` element exists in the template. When querying for the `img` to check the image exists, use the `[slot="header"]` selector, chaining the `get` method for the `img`. `get` starts at the currently selected element and traverses DOM until it finds a match. Since the `HTMLImageElement` has the same attribute, it matches the same node.

```
    cy.get("#root").find('[slot="header"]').get("img").sh\
ould("exist");
```

Continue testing the content by querying for the remaining elements projected into the header, content, and footer slots, this time asserting the DOM nodes contain specific string characters that are expected.

```
    cy.get("#root").find('[slot="header"]').contains("Foo\
d");
    cy.get("#root").find('[slot="content"]').contains("Lo\
rem ipsum");
    cy.get("#root").find('[slot="footer"]').contains("Rea\
d");
```

When you are finished writing the test that verifies that the content is correctly projected into each slot, the test should appear like below.

```
  it("should find the card slot content in light DOM", ()\
 => {
    cy.visit("iframe.html?id=components-card--image-card"\
);
    cy.get("#root").find('[slot="header"]').get("img").sh\
ould("exist");
    cy.get("#root").find('[slot="header"]').contains("Foo\
d");
    cy.get("#root").find('[slot="content"]').contains("Lo\
rem ipsum");
    cy.get("#root").find('[slot="footer"]').contains("Rea\
d");
  });
```

The final test should appear as follows. Once you are finished writing the tests, run the `Card.spec.js` test in the Cypress user interface to verify the integration tests pass.

```
describe("CardComponent", () => {
  it("should find slots in the card", () => {
    cy.visit("iframe.html?id=components-card--image-card"\
);
    cy.get("in-card").shadow().find('slot[name="header"]'\
).should("exist");
    cy.get("in-card").shadow().find('slot[name="content"]\
').should("exist");
    cy.get("in-card").shadow().find('slot[name="footer"]'\
).should("exist");
  });

  it("should find the card slot content in light DOM", ()\
 => {
    cy.visit("iframe.html?id=components-card--image-card"\
);
    cy.get("#root").find('[slot="header"]').get("img").sh\
ould("exist");
    cy.get("#root").find('[slot="header"]').contains("Foo\
d");
    cy.get("#root").find('[slot="content"]').contains("Lo\
rem ipsum");
    cy.get("#root").find('[slot="footer"]').contains("Rea\
d");
  });
});
```

Integration tests are an essential part of any testing strategy for user interface components. You spent time crafting a API for engineers to plop content into a header, content, and footer layout. Writing integration tests ensure that effort doesn't go to waste and protects against breaking changes if another engineer changes the template.

When you are completed with the integration tests, stop the process running cypress in open mode in your Terminal.

Snapshot tests

While integration tests verify the parts of the codebase work together, snapshot tests validate the visual appearance of the component. Unlike integration tests, snapshot tests require no actual coding. In the repository a test framework named Jest is installed, along with a Storybook addon named StoryShots[37]. StoryShots integrates Jest snapshot testing[38] with Storybook.

When you run the snapshot tests, a visual snapshot of each Story is generated in the same directory of the Story in a directory named `snapshots`. These directories are tracked and versioned with git. Whenever an engineer changes CSS, altering the appearance of the component, the snapshot test will fail, generating a diff in the git changes.

Before you run the first snapshot test for `CardComponent`, we need to do some housekeeping.

Delete the file `packages/component/src/Default.stories.js`.

```
rm packages/component/src/Default.stories.js
```

That file came with the starter code for the repository. If you fail to delete this file, the snapshot tests will generate a snapshot for the story. With the boilerplate story out of the picture, generate the snapshot for the Image Card story.

```
yarn test:snapshot
```

Notice the feedback in the Terminal and look in the `packages/component/src/card/snapshots` directory. A new file was generated that is a visual snapshot of the Image Card story.

[37] https://storybook.js.org/addons/@storybook/addon-storyshots
[38] https://jestjs.io/docs/snapshot-testing

Commit your changes in git to complete this section. Now you've completed integration and snapshot tests for `CardComponent` let's review the content of Chapter One.

Summary

In this chapter, you coded the first autonomous custom element using several specifications that comprise Web Components: Shadow DOM, HTML templates, and ES Modules. You used Shadow DOM to provide encapsulation around the component template, isolating component styles and elements from the global scope of Light DOM. CSS variables came in handy when you styled the component because they pierce the shadow boundary unlike styles from a global stylesheet. Slots provide a dynamic template for the component, allowing you to project different content into the header, content, and footer.

`CardComponent` was exported using ES Modules and imported into `Card.stories.js`, a file where Storybook Stories are declared. You scaffolded the `ImageCard` Story, which provided a development environment for coding `CardComponent` and test bed for testing the component with Cypress and StoryShots.

Subsequent chapters follow a similar trajectory. You'll develop several user interface components using custom elements and browser specifications, importing the component `class` into a Story, then use Storybook as a development environment for coding the template, styling, and behavior of the component you're focusing on in each chapter.

This chapter about autonomous custom elements was broad in scope to introduce you to the set of specifications known as Web Components. Later chapters narrow the scope of the material, while reinforcing the concepts you learned in this chapter. If you've made it this far you are prepared to continue onward and learn

about the other two kinds of custom elements: form-associated and customized built-in elements.

Form-associated custom elements

Chapter Two

Introduction

Whether it's a login view, checkout in an e-commerce app, or a table where every cell is editable, there's no shortage of form controls in modern applications. Autonomous custom elements have some problems when working in the context of HTML forms. When a form control is encapsulated by Shadow DOM, like in the case of this input, the form control cannot participate in the parent `HTMLFormElement`. By default, the `HTMLInputElement` should report validity and it's value to a parent `HTMLFormElement`, however, the Shadow DOM boundary blocks communication between the form and input.

```
▼<form>
  ▼<in-textinput required name="username">
    ▼#shadow-root (open)
      ▼<div class="control">
          <input type="text"> == $0
        </div>
        <div class="message"></div>
    </in-textinput>
  </form>
```

When a form control is encapsulated by Shadow DOM, the element:

- Will not be listed under the parent `HTMLFormElement` node list
- Can't be validated alongside other form controls properly
- Won't be included in form submission

When form inputs are not properly configured, accessibility issues can also arise because screen readers expect form controls to have specific relationships with elements that label them. Shadow DOM can break the connection between a label and form control.

Form-associated custom elements solve these problems by introducing the `ElementInternals`[39] interface on the custom element. The `ElementInternals` interface allows the custom element to participate in a form like any typical form control. Native form controls like `HTMLInputElement` and `HTMLTextAreaElement` already employ similar methods found on the `ElementInternals` interface that allow them to register with the `HTMLFormElement` and participate in the form, but if the form control is contained in Shadow DOM, those methods are inaccessible to the form.

When the `formAssociated`[40] property is set to `true` on the custom element `class`, `ElementInternals` is provided to an autonomous custom element by calling `attachInternals()`. It's typical to set another property on the `class` to an instance of `ElementInternals` so you can reference it later. When an autonomous custom element is configured in such a way, it becomes a form-associated custom element.

[39] https://developer.mozilla.org/en-US/docs/Web/API/ElementInternals

[40] https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-face-example

```
export class TextInputComponent extends HTMLElement {
  static formAssociated = true;
  private internals: IElementInternals;
   constructor() {
    super();
    attachShadow(this);
    this.internals = this.attachInternals();
  }
```

`ElementInternals` provides a bunch of utility methods on the custom element `prototype` used to:

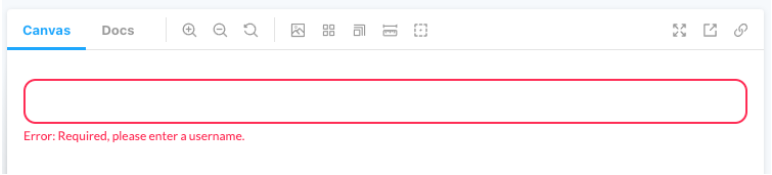
- Validate the form control
- Assign labels for screen readers
- Communicate state used during form submission
- handle accessibility behaviors

The `ElementInternals` interface includes the entire Accessibility Object Model[41], so the custom element can retain all the accessibility characteristics expected from a form control. A full reference of `ElementInternals` interface is provided on MDN.[42].

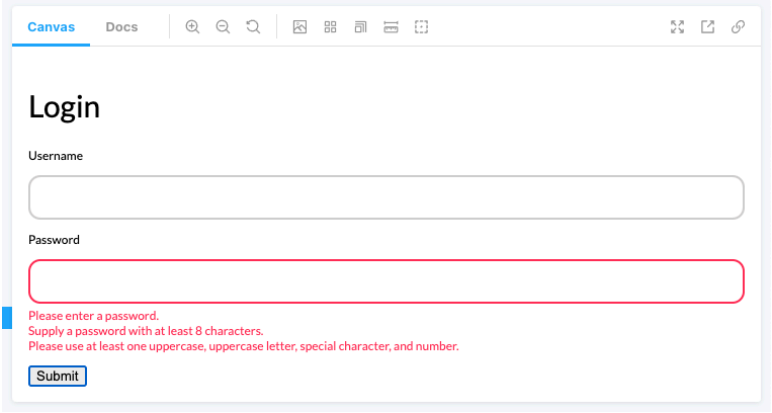
What you will build

In this chapter, you'll code a form control used throughout the application: `TextInputComponent`. This element displays editable text input and a list of validation messages to a user.

[41] https://wicg.github.io/aom/spec/
[42] https://developer.mozilla.org/en-US/docs/Web/API/ElementInternals



You will develop stories to develop each of the `TextInputComponent` states (default, error, and disabled) and a login view that mocks form validation and submission.



By the end of this chapter, each `TextInputComponent` should handle validation and participate in form submission. When the user navigates away from the form control with mouse or keyboard input, blurring the form control, the custom element should validate and display user feedback. If the user attempts to submit the form, the value of each `TextInputComponent` should be included in the `FormData Object` found on the `HTMLFormElement`.

To achieve these goals, `TextInputComponent` has to use several lifecycle hooks provided by `HTMLElement`. Form controls usually provide an interface so that developers can set different properties via attributes, like in the case of making `HTMLInputElement` have a required state. You'll provide a similar interface for `TextInputComponent`, requiring the use of two methods available

to all custom elements:

- `observedAttributes` to declare which attributes should listen for changes
- `attributeChangedCallback` to handle changes to attributes declared in `observedAttributes`
- `connectedCallback` when logic needs to run after the custom element is instantiated

By listening to attributes on `TextInputComponent` you'll be able to pass attribute values to a `HTMLInputElement` embedded in the component Shadow DOM through `attributeChangedCallback`. Another lifecycle hook named `connectedCallback` allows you reliably bind event listeners to child elements in Shadow DOM that don't exist until after the component has been inserted into DOM and is a more reliable space to query for child elements than the `constructor`.

Just like in the previous chapter, these new custom elements will extend from `HTMLElement`, use `ShadowDOM` for encapsulation, and register with the `CustomElementRegistry` in the same way. What sets form-associated custom elements apart from autonomous custom elements is the addition of the `ElementInternals` interface and a new property named `formAssociated`. When `formAssociated` is `true`, the custom element provides a slew of other lifecycle hooks. In this chapter, you'll use:

- `formDisabledCallback` to react to changes in the disabled states of `fieldset`,
- `formStateRestoreCallback` when the user agent makes attempts to restore form control state, and
- `formResetCallback` to reset the value of the form control.

By the end of this chapter, you should be able to code form-associated custom elements for a variety of use-cases.

What you will learn

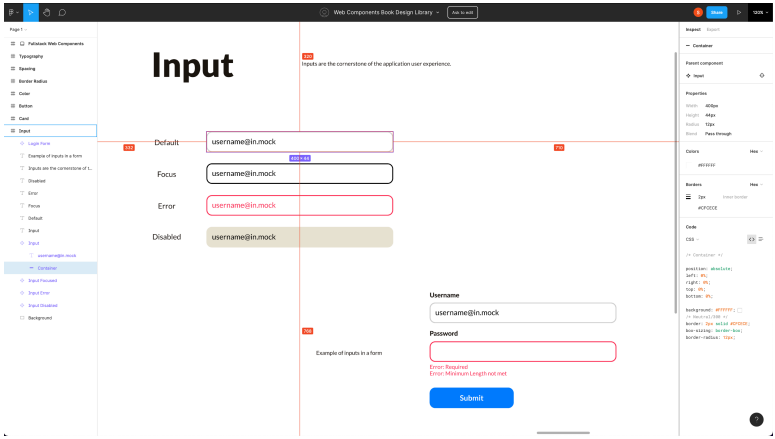
- What is a form-associated custom element?
- What are the benefits of adopting form-associated custom elements?
- What is the lifecycle of a form-associated custom element?
- How do form-associated custom elements participate in form submission and validation?
- How are disabled and error states handled in form-associated custom elements?
- How does styling based on changes to component state work?
- How do custom elements listen for attribute changes?
- How can custom elements reliably interact with children?
- How can custom elements communicate with parent DOM nodes?

Acceptance criteria

- A class called `TextInputComponent` is exported from `packages/component/src/input/TextInput.ts`
- `TextInputComponent` is styled per mockup in Figma
- `TextInputComponent` supports the same attributes of `HTMLInputElement`
- `TextInputComponent` has Stories that display the different states of a form control: default, focus, disabled, and error
- `TextInputComponent` is tested with Cypress and Storyshots

You can review the design mockup for `TextInputComponent` in Figma[43].

[43] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=324%3A101



Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-2 from where you left off at the end of chapter 1.

```
git checkout -b chapter-2
```

Run Storybook:

```
yarn storybook
```

Scaffolding TextInputComponent

The process for scaffolding `TextInputComponent` looks remarkably similar to the `CardComponent` from the previous chapter. You'll

define the custom element, register and export the custom element, import the custom element into a story, then define the story and template. From that point forward, you'll have a development environment for coding the features of the component and a way to test how the component behaves when focused, blurred, and disabled, and how the component displays user feedback during validation.

Make a new directory at `packages/component/src/input`.

```
mkdir -p packages/component/src/input
```

In the input directory, make a new file and name it TextInput.ts.

```
touch packages/component/src/input/TextInput.ts
```

Export a class called `TextInputComponent` that extends from `HTMLElement`, making sure to include a call to `super()`, which is necessary for every custom element, regardless of type. This new `class` should be exported for use in stories. Form-associated custom elements extend from the same `class` as autonomous custom elements, `HTMLElement`, which allows them to employ Shadow DOM.

```
export class TextInputComponent extends HTMLElement {
  constructor() {
    super();
  }
}
```

Next, declare the `ShadowRoot` by calling `this.attachShadow`, leaving the mode `open`. Create a template element and clone the template to the `ShadowRoot` using `template.content.cloneNode(true)`.

```
    const shadowRoot = this.attachShadow({ mode: "open" }\
);
    const template = document.createElement("template");
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
```

After the class declaration, add a line to `TextInput.ts` that registers the custom element with the `CustomElementRegistry`, using the tag name `in-textinput`.

```
customElements.define("in-textinput", TextInputComponent);
```

When you are finished, the scaffolded component should look as follows:

```
export class TextInputComponent extends HTMLElement {
  constructor() {
    super();
    const shadowRoot = this.attachShadow({ mode: "open" }\
);
    const template = document.createElement("template");
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
  }
}

customElements.define("in-textinput", TextInputComponent);
```

Structure the content

`TextInputComponent` should have a `HTMLInputElement` in the component template, mainly because we don't want to duplicate

all the functionality of `HTMLInputElement`. We can't extend `TextInputComponent` from the `HTMLInputElement class` and expect to also leverage form-associated custom elements, which require `HTMLElement`. If `TextInputComponent` extended from the `HTMLInputElement class`, it would be a customized built-in element and lose the ability to incorporate Shadow DOM or become a form-associated custom element.

The `ShadowRoot` should contain two essential DOM nodes:

- The first `HTMLDivElement` with the class name `control` contains the `HTMLInputElement`
- The second `HTMLDivElement` classed `message` is responsible for displaying validation messages

Set the `template.innerHTML` with the following template. Make a container div with a classname `control`. Inside of this div, insert an input and set the `type` attribute to `text`. This ensures the default type will always be `text`.

```
1     template.innerHTML = `
2     <div class="control">
3       <input type="text" />
4     </div>
```

Add another container div directly after the first, this time using the classname `message` to the div. Leave this `HTMLDivElement` empty for now; it will be filled dynamically with the content of each validation message later.

```
    template.innerHTML = `
    <div class="control">
      <input type="text" />
    </div>
    <div class="message"></div>
    `;
```

Looking back at the Figma[44], it appears the structure of the template is just enough to support the input and validation messages, but lacks support for the label. This is intentional because the label must be a sibling of `TextInputComponent`, not a child, so screen readers can effectively analyze the form for accessibility purposes. This behavior replicates the default relationship between form labels and inputs, only now `TextInputComponent` stands in as the input, rather than `HTMLInputElement`. Now that you've declared the template and registered the custom element for `TextInputComponent`, you'll be able to see a plain text input in Storybook, but only after you define the story.

Define the story

In the input directory, make a new file and name it `TextInput.stories.js`.

```
touch packages/component/src/input/TextInput.stories.js
```

Import `TextInputComponent` so you can reference the tag name in each story.

```
import { TextInputComponent } from "./TextInput";
```

[44] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=324%3A101

Declare a default export in Component Story Format, setting the title to `Components/Inputs/TextInput` and component to the tag name `in-textinput`.

```
export default {
  title: "Components/Inputs/TextInput",
  component: "in-textinput",
};
```

Next, declare a `const` named `PrimaryTemplate` and make it equal to a function that returns a string with the template: using the HTML tags for `TextInputComponent`. Wrap the `in-textinput` tags in a `HTMLFormElement`. Import `html` from the lit-html package, so Storybook can properly pass values from Controls to the attributes of the `TextInputComponent`.

Import `html` from the `lit-html` package to satisfy Storybook's dependency on Lit for all Web Component templates.

```
import { html } from "lit-html";
```

Export a `const` named `Primary` and make it equal to `PrimaryTemplate.bind({})`.

```
const PrimaryTemplate = ({}) =>
  html`<form><in-textinput></in-textinput></form>`;
export const Primary = PrimaryTemplate.bind({});
```

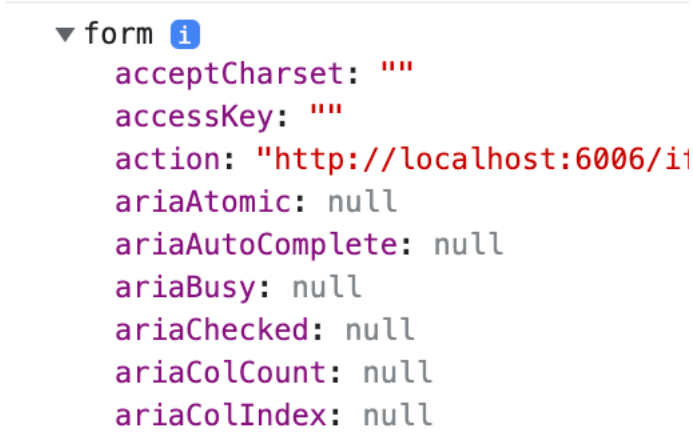
Navigate to the **Inputs** directory in the Storybook sidebar and select **TextInput**, then select **Primary**, and you should see a plain text input in the Canvas.



In the following sections of this chapter, you'll style the input per the mockup in Figma, but for now, let's explore the API that allows the `HTMLInputElement` encapsulated by the `ShadowDOM` to participate in a form.

Form Associated and ElementInternals

To establish why additional API is needed to utilize the `HTMLInputElement` embedded in the autonomous custom element in the context of a HTML form, inspect the DOM using Dev Tools and observe the `HTMLFormElement`. In Chrome, right-click on the input in the Storybook Canvas and select **Inspect**. In the Dev Tools Elements tab, select the form tag, right-click on the tag, and select **Store as global variable** from the dialog. This makes a global variable called `temp1` available in the Dev Tools Console. Type `dir(temp1)` in the Console and press Return. This should print an `Object` in the Console that allows you to inspect the `HTMLFormElement` DOM node.



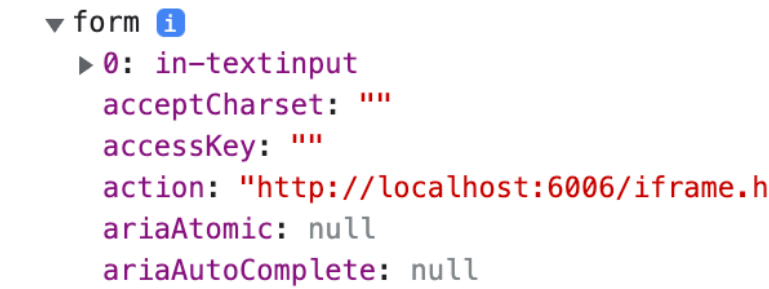
Notice that no input elements are listed in the form element's node list usually located as the first properties of the form `Object`, despite there being an `HTMLInputElement` that is a child of the form. Only elements listed directly on the form `Object` can participate in the form. Since the `HTMLInputElement` is embedded in `ShadowDOM` and not part of the Light DOM like the form, the input cannot participate.

Transforming an autonomous custom element into a form-associated custom element requires minimal additions to the `TextInputComponent` class definition. To make the autonomous custom element a form-associated custom element, declare a new property: `formAssociated` prior to the constructor.

```
1   static formAssociated = true;
```

The `static` property `formAssociated` should be equal to `true`. This property is used so the browser can feature detect, resolving the custom element is a form-associated custom element.

After declaring `formAssociated`, inspect the DOM again, and you'll find the custom element has been added to the `HTMLFormElement` DOM node list and has an index of `0`.



Even though the custom element is now eligible to participate in the form, we still need to give the element API consistent with a typical form control like `HTMLInputElement`. `attachInternals` is a method that was added to the `HTMLElement` spec for that purpose. A call to the `attachInternals` method grants access to methods on the `ElementInternals` interface for the custom element. `ElementInternals` gives a form control the ability to communicate value and validity with the form.

Before the closing bracket of the `constructor`, set `this.internals` equal to `this.attachInternals()`.

```
1     this.internals = this.attachInternals();
```

Once you do that, TypeScript may throw two errors: `Property 'attachInternals' does not exist on type 'TextInputComponent'.` and `Property 'internals' does not exist on type 'TextInputComponent'.`. That's because the type definitions exported from `lib.dom.d.ts` do not currently support `ElementInternals` interface. `lib.dom.d.ts` is where TypeScript stores all the type definitions related to DOM APIs. To fix this type discrepancy, a type definition is included in the project repository in the file /types/lib.elementInternals.d.ts. This static type definition will not be necessary when the TypeScript DOM library properly exports the type definition for `ElementInternals`.

```
import { ElementInternals } from "types/lib.elementIntern\
als";
```

Before the constructor, declare the type for a `private` property called `internals` and set the type to `ElementInternals`. Declare the type for the `private` property `attachInternals` as `() => ElementInternals`.

```
  private attachInternals: () => ElementInternals;
  private internals: ElementInternals;
```

Setting `formAssociated` allows the browser to detect that the custom element is a form-associated custom element, while `ElementInternals` grants several methods to the prototype of the `TextInputComponent` class. These methods are needed for the custom element to communicate with the form. Inspect the contents of `lib.elementInternals.d.ts` to become familiar with the methods you'll be working with throughout this chapter.

Handling Validation

Now the `TextInputComponent` can communicate with a form, we should expose some of the methods used during form validation on the component `class`. Validation state is stored on `this.internals`, which is both `private` and possibly not as intuitive for engineers to access validity state. If validity state were to remain on `internals`, it would mean `TextInputComponent` won't behave like a typical form element which stores this state on the root of the `Object`. We don't necessarily want developers to reference `input.internals.validity` when looping over a form's inputs to check validity. Accessing validity state from `input.validity` is much more preferable.

Create methods on the component class that expose the methods on the `ElementInternals` interface used for validation. First, make a method called `checkValidity` that returns `this.internals.checkValidity()`. This method checks for any constraints that have been declared, and when the check fails for any of those constraints, triggers the browser to emit an `invalid` event, and then returns `false`.

```
  checkValidity() {
    return this.internals.checkValidity();
  }
```

`reportValidity` is very similar to `checkValidity`, except that it also sends any `validationMessage` to the user agent for display. Add the `reportValidity` method to the `TextInputComponent` class, and make sure it returns `this.internals.reportValidity()`.

```
  reportValidity() {
    return this.internals.reportValidity();
  }
```

`validity` returns a `ValidationState Object` that includes properties that reflect common validation errors, such as `tooShort`, `valueMissing`, and `patternMismatch`. This property is read-only. `ValidationState` is updated on calls to `setValidity` and `reportValidity`. Add the `validity` getter to the `class`, and make sure it returns `this.internals.validity`.

```
  get validity() {
    return this.internals.validity;
  }
```

The next validation-related getter named `validationMessages` returns messages associated with each invalid condition of the form control. Add the `validationMessages` getter to the class, and make sure it returns `this.internals.validationMessages`.

```
  get validationMessage() {
    return this.internals.validationMessage;
  }
```

`ElementInternals` includes a method called `setValidity` that does most of the heavy lifting during form validation. This method takes three arguments: `flags`, `message`, and `anchor`:

- `flags` is an `Object` with properties that correspond to the validation flags the browser can interpret
- `message` allows the engineer to pass a custom human-readable `string` that can be used to display feedback for the end-user
- `anchor` is set to the `HTMLElement` the engineer wishes to display the user feedback stored in `message`

```
  setValidity(
    flags: ValidityStateFlags,
    message?: string,
    anchor?: HTMLElement
  ): void {
    this.internals.setValidity(flags, message, anchor);
  }
```

Whenever validation is triggered on the form control, usually when the user changes the value or blurs the input, you can write logic that validates the current value of the form control against some condition.

Consider the scenario of the required form control. The user focuses the input, then immediately navigates away without entering text into the field, effectively blurring the input. You could write a condition like the following that checks if the value of the text input has any content and then triggers validation on the `ElementInternals` interface by setting a validation error with the key `valueMissing` equal to `true` and a human-readable message.

```
if (input.value && input.value.length > 0) {
  input.setValidity(
    {
      valueMissing: true,
    },
    "Error: This field is required, Please type in a valu\
e."
  );
}
```

There is an important distinction between the first and second arguments of `setValidity`. The `ValidityStateFlags` on the `Object` in the first argument communicates to the user agent what is invalid, while the message composed from a `String` in the second argument provides user feedback for the user to acknowledge the error and take action. The message `Error: This field is required, Please type in a value.` notifies the user there is a problem, prompting them to type something into the text field as a solution.

Two questions arise from the above example. Where does validation logic like the above belong in source code? How do you get the message to display in UI? For now, let's tackle the answer to this first question. Later in this chapter, we'll dive into the second.

Where does validation logic belong in the source code?

The context of the `TextInputComponent` provides a convenient place to listen for events on the `HTMLInputElement` and provide a callback to validate the form control. Suppose somewhere in the component, you listened for the blur event on the input and triggered validation on a method called `onValidate`.

```
this.$input.addEventListener("blur", this.onValidate);
```

In the `onValidate` callback, you could check if the element has a required attribute and the validity of the input. When the condition returns false, call `setValidity` with the `valueMissing` flag because it matches this use-case and viable user feedback in the second argument.

```
if (
  this.getAttribute("required") &&
  this.$input.value &&
  this.$input.value.length > 0
) {
  this.setValidity(
    {
      valueMissing: true,
    },
    "Error: This field is required, Please type in a valu\
e."
  );
}
```

This pattern seems like it would work but has some limitations. With the conditions hard-coded in the component itself, developers using the component can't override them. In other words, validation isn't configurable based on different use-cases. This is especially important considering that one of the properties on `ValidityStateFlags` is `customError`. Developers can call `setValidity` with a completely custom validation error that isn't covered by the available flags. How would you support dynamic use-cases when the conditions are hard-coded?

One solution is to put the conditions elsewhere, like the application logic as opposed to the component logic. That way, developers using the component have full control over the validations of

each form control. The component no longer prescribes validation, but the component could still handle calls to `setValidity`. Let's explore how this could be architected by establishing a pattern for validations that are declared in the story, then passed to the component by setting a property on the component.

Sketching the Validator type definition

Start by sketching out the API for how engineers would handle each validation in application logic. Make a new file at `packages/component/src/input/validator.ts`.

```
touch packages/component/src/input/validator.ts
```

Define the type definition for an interface called `Validator` here and export the new `type` from the file so it can be used elsewhere.

```
export type Validator = {
  validations: {
    flag: ValidityStateFlags;
    condition: (elem: HTMLElement) => boolean;
    message?: string;
  }[];
};
```

`Validator` is an `Object` so that the API can possibly expand over time. For now, `Validator` contains one property: `validations`. A single form control could have multiple validations, so `validations` is an `Array` of `Object`. Each `Object` specifies the flag(s) from `ValidityStateFlags` associated with the validation on the `flag` property. A human-readable message is set on the `message` property. Both `flag` and `message` can be passed directly to `setValidity`. The third property called `condition` is equal to a `function` that takes an element as an argument and returns a `boolean`. We should assume

the element passed into the first argument is the form control we're validating against.

Mocking the Validators in Storybook

In `TextInput.stories.js`, declare a new `const` named `validators`. Make `validators` equal to an `Object` with a property `username`, which is equal to another `Object`, the `Validator`.

```
const validators = {
  username: {
    validations: [
      {
```

The `username Validator` should have a property named `validations` that is equal to an `Array`. Specify the Objects in this `Array` for the input used to enter a username. The validation below tests if the input `required` property is truthy and the `value` has length.

```
const validators = {
  username: {
    validations: [
      {
        flag: { valueMissing: true },
        message: "Error: Required",
        condition: (input) => input.required && input.val\
ue.length <= 0,
      },
    ],
  },
};
```

For validation to work, you'll need to store it on the component instance. To do that, give the `in-textinput` a new attribute called `name` and set the `name` attribute to `username`.

```
<form><in-textinput name="username"></in-textinput></form>
```

When you query for DOM here in the template function, you'll find that the `in-textinput` isn't available yet. That's because LitHTML `html` function has to compile the template, so any queries have to be made behind a call to `setTimeout`. Using `setTimeout`, query for the custom element and set a property called `$validator` on the element, making it equal to `validators["username"]`. Setting properties like this on a custom element is one way for application logic to communicate with the component.

```
const PrimaryTemplate = ({}) => {
  setTimeout(() => {
    const input = document.querySelector(`[name="username\
"]`);
    input.$validator = validators["username"];
  }, 0);
  return html`<form><in-textinput name="username"></in-te\
xtinput></form>`;
};
```

Uptaking the Validators in TextInput

Back in `TextInput.ts`, you'll need to make some changes to uptake the validations. Import `Validator` from `validator.ts`, and declare a new `$validator` property on the `class`, making sure it's `public` and type defined as `Validator`.

```
  public $validator: Validator;
```

Declare a new getter on the `class` called `$input` and return the embedded `HTMLInputElement`.

```
  get $input(): HTMLInputElement {
    return this.shadowRoot.querySelector("input");
  }
```

Source of truth

Before we move forward, there is an important distinction to make between the custom element and the input. The form-associated custom element is where communication happens between the component and the form, while the input is where all user interaction typically takes place. When a user clicks the input or navigates there via the keyboard, focusing it, they are interacting with the input, not the custom element.

This poses a problem. Where is the source of truth for component state? So far, you've exposed methods on the `class` that suggests the source of truth is derived from properties and attributes on the custom element, but if you were to set a `required` attribute on the custom element, how does the input interpret this change to the required state? Conversely, when the input value updates, how does the custom element report the value to the outside world?

Part of the problem is that form controls usually expose getters that expose the state of required, disabled, the value, and so on, except the only stable way to set the state on the form control is via attributes. To make a `HTMLInputElement` required, for instance, you have to set the `required` attribute on the input.

```
<input type="text" required />
```

When developing reusable UI components, the outward-facing API of the component matters. If we expose the component state only through getters and setters, it would be awkward for engineers used to setting attributes on form controls to set state. Consider a problem we've already introduced concerning the design of the `Validator`. In the condition declared in `TextInput.stories.js`, the first argument `input` checks if the `input` has a property `required` and validates if the input has a value by assessing `input.value.length`. But what is `input`? Is `input` referencing the `TextInputComponent` or the embedded `HTMLInputElement`?

```
validations: [
    {
      flag: { valueMissing: true },
      message: "Error: Required",
      condition: (input) => input.required && input.value\
.length <= 0,
    },
],
```

One solution is to treat the `HTMLInputElement` as the single source of truth for component state and essentially use `TextInputComponent` as a pass-thru, exposing properties of the `HTMLInputElement` with getters on the component `class` and using setters that change the state of the `HTMLInputElement`. That way, `TextInputComponent` exposes an outward-facing API. Let's see how this works in action. To achieve this we need a way to set attributes on `in-textinput` and have the component react to changes to those attributes, similar to how a typical input element behaves.

Listening for attribute changes

Luckily, custom elements provide an API for observing changes to attributes and reacting to those changes. Every custom element

has access to a `static` getter called `observedAttributes`. Using `observedAttributes`, you can establish which attributes the component listens to for changes. For performance sake, you have to whitelist each attribute that should be observed.

Just below the `constructor` in the `TextInputComponent class`, declare the `observedAttributes` getter. `observedAttributes` returns an `Array` of `String`, each `String` corresponding to the name of the attribute that should be observed. Add `required` and `value` to the `Array`. You're declaring `observedAttributes` so that developers can set the `value` and `required` attributes on `TextInputComponent` because those are the properties the condition in the `Validator` is referencing to validate whether or not the field has value. You'll add more attributes that should be observed later, but for now, `required` and `value` should suffice.

```
1   static get observedAttributes() {
2     return ["required", "value"];
3   }
```

After declaring which attributes should be observed, a second method named `attributeChangedCallback` handles attribute changes. `attributeChangedCallback` is another method available to all custom elements that must be used in conjunction with `observedAttributes`.

`attributeChangedCallback` has three arguments, `name`, `previousValue`, and `currentValue` that pass the name of the changed attribute, the previous value of the attribute, and the current value of the attribute, respectively. For the sake of brevity, we'll call these arguments `name`, `prev`, and `next`. You could strictly type define the arguments and what the method returns, but for now we'll ignore the types for this method. `prev` and `next` could have many different types, so type defining these attributes anything other than `any` can become tricky. `name` will most certainly always be of type `string`.

```
  attributeChangedCallback(name, prev, next) {
```

Looking at the `attributeChangedCallback`, it appears we need a way to do something based on the value of `name` passed through the first argument of `attributeChangedCallback`. A typical convention is to use a `switch` statement.

```
    switch (name) {
      case "value":
        this.value = next;
        break;
      case "required":
        this.required = next;
        break;
    }
```

Add the `switch` to `attributeChangedCallback`, passing in the `name`. Set up `case` for `value` and `required`. If you try to set `this.value` and `this.required` based on the value of `next`, you'll be prompted with errors the `value` and `required` properties do not exist, so let's add them. When you are finished `attributeChangedCallback` should appear like the example below.

```
 attributeChangedCallback(name, prev, next) {
    switch (name) {
      case "value":
        this.value = next;
        break;
      case "required":
        this.required = next;
        break;
    }
 }
```

Using getters and setters to communicate state

Existing form controls like `HTMLInputElement` and `HTMLTextAreaElement` set their own state through attributes, and in some cases, properties. `value` and `disabled` can reliably be set through properties on `HTMLInputElement`, but there is no equivalent setter for `required`; required state must be set through an attribute. In this section, you'll wrangle these odd behaviors by adding getters and setters to `TextInputComponent class`.

Earlier, we tried to set `value` in the `attributeChangedCallback`, but it failed because there currently isn't a `value` setter, so let's add one. A setter method in JavaScript starts with the `set` keyboard, followed by the name of the method: `value`. The first argument of this method is the value that should be set. The setter passes the value argument to the `HTMLInputElement` by setting that element's `value` property.

```
1   set value(value: string) {
2     this.$input.value = value;
3   }
```

To allow the `Validator` access to the current `value` of the `HTMLInputElement`, make a getter on the `TextInputComponent class`. Getters in JavaScript start with the `get` keyword, followed by the method name: `value`. There's no arguments here, but you can set the return value as `string`, then in the curly brackets of the method, `return` the `HTMLInputElement` current value state: `this.$input.value`.

```
1   get value(): string {
2     return this.$input.value;
3   }
```

Repeat the same process for getters and setters for the required state, although this time, you'll have to set state of the `HTMLInputElement` by setting its `required` attribute. When passing the required state from the `TextInputComponent` attribute through `attributesChangedCallback`, the type of the required state will be `string`, because that's usually the type of anything passed through attributes. This is a problem because when coding the getter for required, you can reliably access the required state of the `HTMLInputElement` through the `required` property, which returns the required value with the type `boolean`.

Add the `required` getter to `TextInputComponent`.

```
  get required(): boolean {
    return this.$input.required;
  }
```

In TypeScript, getter and setter types must match, so the setter has to accept both a `string` and `boolean` in the first argument. Code the `required` setter, adding conditions that check the `value` and either call `setAttribute` or `removeAttribute` on the `HTMLInputElement`.

```
  set required(value: boolean | string) {
    if (value === "true" || value === true) {
      this.$input.setAttribute("required", "true");
    }
    if (value === "false" || value === false) {
      this.$input.removeAttribute("required");
    }
  }
```

Once you've completed the `value` and `required` getters and setters, return to TextInput.stories.js and add the `required` attribute to `in-textinput` in the story template.

You'd expect to see a `required` attribute set on the `HTMLInputElement`. However, if you use Dev Tools to observe

the changes in DOM, you'll notice the `required` attribute is still missing.

```
▼<form>
  ▼<in-textinput required name="username">
    ▼#shadow-root (open)
      ▼<div class="control">
          <input type="text"> == $0
        </div>
        <div class="message"></div>
    </in-textinput>
  </form>
```

This is something we need to fix since `TextInputComponent` relies on the state of `HTMLInputElement`, but what is the problem?

Passing attribute values to children

It's likely the `HTMLInputElement` hasn't received a call to `setAttribute` because it doesn't exist yet. Remember, the `HTMLInputElement` is encapsulated by Shadow DOM. When `TextInputComponent` instantiates, `attributeChangedCallback` fires almost immediately, but the browser hasn't yet fully parsed all the nodes in the `ShadowDOM`.

To handle situations like this, custom elements include another lifecycle method named `connectedCallback` that runs when the custom element is added to DOM. `connectedCallback` offers a space where you can reliably query DOM nodes in the custom element `ShadowRoot`.

So, what does this mean for setting attributes reliably when you need to pass the value of the attribute from parent to child node? This means you need to store the values of the attribute somehow and defer setting the child node's attributes to `connectedCallback`. Eventually, you're going to handle many more different attributes

in `attributeChangedCallback`, so create a new property on the `TextInputComponent` class called `$attr` and make it equal to an empty `Object`.

```
  private $attr = {};
```

In the first line of `attributeChangedCallback`, set properties on `this.$attr` with whatever passes through the callback.

```
    this.$attr[name] = next;
```

Before or after the declaration of `attributeChangedCallback` in `TextInputComponent`, make a new method called `connectedCallback`.

```
connectedCallback() {
    for (let prop in this.$attr) {
      this.$input.setAttribute(prop, this.$attr[prop]);
    }
}
```

Inside the method implementation, loop over the properties on `this.$attr` and call `setAttribute` on the `HTMLInputElement` for every attribute currently stored in the `$attr Object`.

Inspect the DOM again in Storybook and notice the `required` attribute is now properly set on `HTMLInputElement`.

```
▼<form>
  ▼<in-textinput required name="username">
    ▼#shadow-root (open)
      ▼<div class="control">
          <input type="text" required>
        </div>
        <div class="message"></div>
    </in-textinput>
  </form>
```

In this section, you learned about `connectedCallback`, a method that fires whenever the custom element is added to DOM. Combined with `attributeChangedCallback`, you developed a means to pass the values of attributes from parent to child whenever the custom element is instantiated, while getters and setters declared on the `TextInputComponent class` handle changing the value whenever it changes during the lifespan of the component. `connectedCallback` and `attributeChangedCallback` are lifecycle methods available on all custom elements. Later in this chapter, you'll learn about lifecycle methods available only to form-associated custom elements.

Handling validation and child events

Since `TextInputComponent` can reliably handle state for required and value, we can finish setting up the `Validator`. To progress, we should consider when validation occurs and clarify some expectations around validation.

- Validation only happens client-side in this application, which is why we need a robust client-based API for validating form controls.
- Validation should happen on blur, i.e., when the user navigates away from the form control, not while the user is interacting with the form control.

- Validation should occur when the component initializes. Otherwise, if the user submits the form without interacting with a form control, the form will be valid and allow submission.

Form controls need to register themselves with their parent `HTMLFormElement` in the `connectedCallback` so that the form can start from an invalid state where applicable. We don't necessarily want to show the UI in an invalid state on initialization, though; any visible user feedback should only display after the form control is blurred in an invalid state. These are not necessarily recommendations for how validation should be handled in every application, but the above describes how the application we're building behaves.

To get started wiring up validation in `TextInputComponent`, add a new method to the `class` named `onValidate`. This method takes an argument `showError`, a `boolean` that determines whether or not to display user feedback upon validation.

```
1   onValidate(showError: boolean) {
```

We could put all the logic here for validation, but other form controls would have to duplicate the code. To keep the implementation DRY (Don't Repeat Yourself), export a new `function` from `validator.ts` named `validate`. This `function` takes two arguments:

- `elem` refers to the form control
- `showError` is the option we're going to pass from `onValidate` in `TextInputComponent` that either shows or hides user feedback

```
1  export function validate(elem: any, showError: boolean) {
```

Import `validate` in `TextInput.ts` and call the `function` in the `onValidate` method, passing `this` into the first argument and `showError` in the second argument. `this`, in this context, means the instance of `TextInputComponent`.

```
  onValidate(showError: boolean) {
    validate(this, showError);
  }
```

Per the component requirements, `onValidate` needs to run on load and after the user navigates away from the form control, on blur. In the `connectedCallback`, call `onValidate` passing in `false` as the first argument. This means error messaging won't be shown initially, but validity state can still be passed to the form element. Add an event handler for the `onblur` event on the input element, this time calling `onValidate` with the first argument set to `true` so validation messages can display properly.

```
  connectedCallback() {
    this.$input.onblur = () => {
      this.onValidate(true);
    };
    for (let prop in this.$attr) {
      this.$input.setAttribute(prop, this.$attr[prop]);
    }
    this.onValidate(false);
  }
```

That's it for now in `TextInputComponent`. You'll return later, but now, let's focus on handling validation in the reusable `validate function` in `validator.ts`.

Changing validation state

Before you add logic to the `validate function`, consider the algorithm you'll have to write. Looking back at the implementation in `TextInput.stories.js`, a property called `$validator` is set on the element. `$validator` is equal to an `Object` that has a property called `validations`: a `Validator Array` that you declared earlier.

In the `validate` function, you'll need to check if `$validator` exists, and if it doesn't, short-circuit the function so that the rest of the logic doesn't run. Another good check is to make sure `validations` exists on the `$validator Object`. This conditional is mainly for instances of the component where validation is not necessary or to prevent errors when developers are implementing other form controls that expose a similar API as `TextInputComponent` or haven't declared the validations yet in application source code.

```
1   if (!elem.$validator || !elem.$validator.validations) {
2     return;
3   }
```

Since the `Validator` supports many validations, the `Array` has to be looped over to check if each `condition` returns `true`. `condition` is a `function` that takes a single argument, the element currently being validated: `elem`. Call `forEach` on `elem.$validator.validations`, and use the callback to loop over each validation in the `Array`.

```
1   elem.$validator.validations.forEach((validator) => {
2     if (validator.condition(elem)) {
3       elem.setValidity(validator.flag, validator.message);
```

When the `condition` returns `true`, make a call to `setValidity`, passing in the `flag` and `message` declared on `validator` to notify the form of an invalid form control.

That takes care of invalid form controls, but how do you notify the form when the form control is valid? You'd need a way to check that no validations met the condition in the `forEach`. One way to handle this is to populate an `Array` with validations that meet the condition. Since `forEach` is blocking, you can check the `length` of the `Array` after the loop. If the `Array` is empty, call `setValidity` with an empty `Object` to notify the form that this form control is valid.

Make a new `Array` named `activeValidators`, and make it equal to an empty `Array`.

```
const activeValidators = [];

elem.$validator.validations.forEach((validator) => {
  if (validator.condition(elem)) {
    elem.setValidity(validator.flag, validator.message);
    activeValidators.push(validator);
  }
});

if (!activeValidators.length) {
  elem.setValidity({});
}
```

In the `forEach`, call `push` on `activeValidators` whenever a `condition` returns truthy to populate the `Array`. After the `forEach`, use another `if` statement to check the `length` of the `activeValidators Array`. When the `length` is falsey, call `setValidity` with an empty `Object`. The absence of any flags on the first argument of `setValidity` tells the `HTMLFormElement` this form control is valid.

```
elem.setValidity({});
```

To test these changes, you'll need a way to listen for changes in validation state and mock an event listener in Storybook.

`HTMLFormElement` doesn't provide hooks into listening for validation state changes, even though under the hood `ValidityStateFlags` state is changing whenever `setValidity` is called. To solve this problem, you can make a `CustomEvent` that bubbles up to the `HTMLFormElement`. Then you can listen to validation changes on the form and mock the event listener in Storybook.

At the very end of the `validate` function, dispatch a `CustomEvent` with the name `validate`, setting the option `bubbles` to `true` so the event bubbles up to the `HTMLFormElement`.

```
1   elem.dispatchEvent(new CustomEvent("validate", { bubble\
2 s: true }));
```

Storybook renders Web Components with LitHTML by default. That's why the template in `TextInput.stories.js` calls the `html function` exported from the `lit-html` package. `html` has some hidden powers we can tap into when mocking the template, specifically the ability to listen for events using a template syntax where you can also pass in an event listener callback.

```
1   return html`<form @validate="${onValidate}">
```

Before you can attach the callback in `TextInput.stories.js`, you have to declare the callback function via Storybook. Locate the place where `Primary` is declared, and immediately after it, set the `Primary.args` to an `Object`. Pass in the `validators` you mocked earlier as one property on the `args Object`, then declare a property called `onValidate`.

```
Primary.args = {
  validators,
  onValidate: (ev) => {
    if (!document.querySelector(`[name="username"]`).vali\
dity.valid) {
      console.warn("INVALID");
    } else {
      console.log("VALID");
    }
  },
```

In the `onValidate` callback, query for the `TextInputComponent` using `document.querySelector` with the attribute you added earlier: `[name="username"]` , and using an `if` statement, check the `validity.valid`. When `valid` is falsey, log `"INVALID"`; otherwise, log `"VALID"`.

Inject `onValidate` and `validations` into `PrimaryTemplate`. Any property on `args` can be injected like this into the template in Storybook.

```
const PrimaryTemplate = ({ onValidate, validators }) => {
```

Bind the `validate` event to the callback using LitHTML `@` attribute syntax. The `@` automatically binds an event listener on the form element, passing in `onValidate` as the callback `function` typically set on `addEventListener`. `@` is one of LitHTML features and not part of the Web Components specifications.

```
  return html`<form @validate="${onValidate}">
```

In Storybook, click on the input and type the Tab key on the keyboard. Observe in the Dev Tools console that `INVALID` is logged. Focus the input again, and type something into the form control. After blurring the input again observe `VALID` is logged. If you do not

see any feedback in the Dev Tools Console, ensure the JavaScript context is set to the storybook-preview-iframe.

While you still have to provide visual feedback for the end-user, you've hooked up everything needed to start validating `TextInputComponent` against user input. `validator` was declared in a separate file because we are anticipating reusing this `function` in other form controls. Abstracting the declaration of each `Validator` to `TextInput.stories.js` simulates real-world usage of the validations. Storybook is providing an environment were you mock components similar to how they would be used in an application. Much later in Part Three of this book, you'll declare validators for username and password fields in the application in the same way. In this section you also wired up an event listener to trigger validation on blur of the input element. The `validator function` also dispatches an event to the nearest `HTMLFormElement` to notify it of the custom element's validity. The form-associated custom element is beginning to participate in the HTML form.

User feedback

Now that you have a test bed where validation state can be verified during development, it's a good time to handle user feedback. According to the mockups in Figma, whenever there is an error associated with a form control:

- The input border should change color to a shade of red
- Messages that instruct the user how to validate the form control are displayed below the input

So far in the `validate function`, you've called `setValidity` whenever the user blurs the form control. Perhaps this is also a good place to handle user feedback. However, since `validate` is a reusable function that could be used in other components, it means any

elements referenced here must be found in all of those custom elements. In `TextInputComponent` template, two elements are used to provide user feedback: the input (`elem.$input`) and a div with the class `message`.

In `validator.ts`, declare a new `const` named `messageElem`. Use `querySelector` to target the div with the class `message` in the `ShadowRoot` of the element.

```
  const messageElem = elem.shadowRoot.querySelector(".mes\
sage");
```

To prevent errors when a form control doesn't include this element, be sure to check if `messageElement` is truthy. Whenever `validate` is called, empty the `messageElement` of content by setting the `innerHTML` to an empty string. This is necessary because existing messages could have accumulated in the element, and in the case of the form control being valid, we need a way to clear them.

```
  if (messageElem) {
    messageElem.innerHTML = "";
  }
```

In the `forEach`, whenever `validator.condition(elem)` is truthy, include some logic to show the error state when the second argument `showError` is `true`. Again, check for the elements before interacting with them. Add a class to the input named `error` and create a new div, set its content to whatever is in `validator.message`, and call `appendChild` on the `messageElem` to append the message.

```
      if (showError) {
        if (elem.$input) {
          elem.$input.classList.add("error");
        }
        if (messageElem) {
          const div = document.createElement("div");
          div.innerHTML = validator.message;
          messageElem.appendChild(div);
        }
      }
```

When the form control is valid, remove the `error` class from the input element and reset the content of `messageElem` by setting the `innerHTML` to an empty string. This has the effect of clearing the error state.

```
  if (!activeValidators.length) {
    elem.setValidity({});
    if (elem.$input) {
      elem.$input.classList.remove("error");
    }
    if (messageElem) {
      messageElem.innerHTML = "";
    }
  }
```

Test out the changes in Storybook. Click on the input and immediately blur, either by clicking outside the input or pressing Tab on the keyboard. You should notice the message "**Error: Required**" below the input. Focus the input, and this time enter a value, and then blur the input, causing the message to disappear. Repeat this process, inspecting the DOM in Dev Tools to ensure that the `error` class is added and removed from DOM at the appropriate times.

Pretty soon, you'll style the input with CSS like the mockup in Figma, and the look and feel will get a substantial improvement.

Before you engage in styling, there's one more state you'll need to handle in `TextInputComponent` whenever the input is disabled.

Handling Disabled State

Form controls can be disabled by setting the `disabled` attribute on the element, or when the element is a descendant of a `<fieldset>`, it should inherit its disabled state. Form-associated custom elements include a lifecycle hook for this specific use-case called `formDisabledCallback`. Whenever `TextInputComponent` is a child to a disabled fieldset in DOM, this callback will be activated. Whenever `formDisabledCallback` is called, the first argument contains the disabled state. Pass this state to the `TextInputComponent`.

```
  formDisabledCallback(disabled) {
    this.disabled = disabled;
  }
```

TypeScript should throw an error because you haven't yet established a setter for the `TextInputComponent` called `disabled`. This setter looks a lot like `required` from before, syncing the child input element with the current value of the `disabled` attribute on the instance of `TextInputComponent`.

```
  set disabled(value: boolean | string) {
    if (value === "true" || value === true) {
      this.$input.setAttribute("disabled", "true");
    }
    if (value === "false" || value == false) {
      this.$input.removeAttribute("disabled");
    }
  }
```

Declare a getter that returns the disabled state from the `HTMLInputElement`.

```
  get disabled() {
    return this.$input.disabled;
  }
```

Add the `disabled` attribute to the `observedAttributes Array`, and then include a `case` in the `switch` statement found in `attributeChangedCallback` that calls the `disabled` setter whenever the `disabled` attribute changes on `TextInputComponent`, making sure to also add the `disabled` to the `Array` returned in `observedAttributes`.

```
      case "disabled":
        this.disabled = next;
        break;
```

That's all there is to giving `TextInputComponent` a disabled state. When a developer sets the `disabled` attribute on `TextInputComponent`, the `disabled` setter will update the `HTMLInputElement` disabled state. Developers can read the disabled state with the `disabled` getter. Whenever a parent fieldset is disabled, the `TextInputComponent` will use the `formDisabledCallback` to set the disabled state of the component via the setter.

Make stories

So far, you've developed the `TextInputComponent` solely using the `Primary` story declared in `TextInput.stories.js`, but during development, two other distinct visual states have appeared: error and disabled.

If you were to begin styling the component with CSS, you would have to configure the story in various ways to trigger the error or disabled state. Disabled, for instance, requires the `disabled` attribute set on `in-textinput` in the template, while error involves mocking user interaction (focus and blur). It's much easier to maintain different stories for each interaction state than configure one story to handle many interaction patterns.

In `TextInput.stories.js` make a new story where the disabled state will be visible whenever someone navigates to the story in Storybook. Start by declaring a `const` named `DisabledTemplate`, a `function` that returns a LitHTML template that sets the `disabled` attribute. Then export a story called `Disabled`, making it equal to `DisabledTemplate.bind({})`.

```
const DisabledTemplate = ({}) =>
  html`<in-textinput
    value="disabled input"
    disabled
    name="test-input"
  ></in-textinput>`;

export const Disabled = DisabledTemplate.bind({});
DisabledTemplate.args = {};
```

Observe the new `Disabled` story appears in the Storybook sidebar, displaying the input with the default browser styling associated with the disabled state.

Using the same format, declare another `const` named `ErrorTemplate`. The template is very similar to `PrimaryTemplate`. This time, however, precede the `return` with a call to `setTimeout`. In the `setTimeout` callback, query for the input, set the `$validator` property to the available `username` validator, then call `focus` and `blur` on the element. This should simulate the action of a user interacting with the component and, as a result, display the

validation message below the input. Make sure to call `setTimeout` with `0` in the second argument, to defer the `setTimeout` callback until after the `LitHTML` template compiles.

```
const ErrorTemplate = ({}) => {
  setTimeout(() => {
    const input = document.querySelector(`[name="username\
"]`);
    input.$validator = validators["username"];
    input.focus();
    input.blur();
  }, 0);
  return html` <in-textinput
    type="text"
    id="username"
    name="username"
    required
    class="form-control"
  ></in-textinput>`;
};

export const Error = ErrorTemplate.bind({});
ErrorTemplate.args = {};
```

To be able to focus and blur the input element via `TextInputComponent`, you'll need to circle back to `TextInput.ts` and add two methods that pass the `function` calls to the child input element.

```
focus() {
  this.$input.focus();
}
blur() {
  this.$input.blur();
}
```

Observe a new `Error` story appears in the Storybook sidebar and now the component displays `Error: Required` and has the `error` CSS class set on the input element. Verify each story is displaying the correct interaction states before styling.

Styling with CSS variables

Now you have three distinct stories to style from: `Primary`, `Error`, and `Disabled`. Reference the mockup in Figma[45] and style each of the interaction states using CSS variables from the global stylesheet.

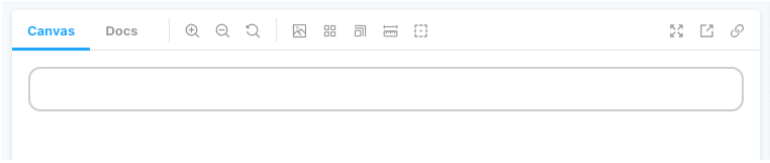
Style the Primary story

Navigate to the `Primary` story in Storybook. In the `constructor` of `TextInputComponent`, make a new `<style>` tag at the start of the template string that sets the template.

Custom elements by default have a `position` of `inline`, and `HTMLInputElement` is defaulted to `inline-block`. However, most designs in the application do not show inputs displayed inline. Instead, the mockups seem to indicate that form controls should have `display` set to `block`, which makes them behave more like a div. Set the `font-family` and `font-size` to establish a baseline for elements in the `TextInputComponent ShadowDOM`.

[45] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=324%3A101

```
:host {
  display: block;
  font-family: var(--font-default);
  font-size: var(--font-body-sm);
}
```



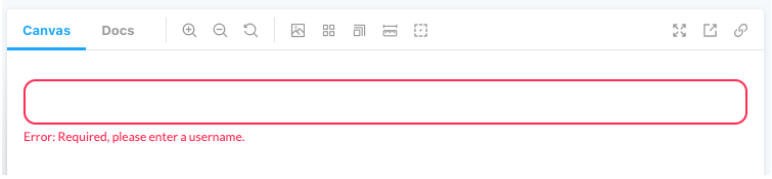
Style the input element, establishing the `height` and `width`. Style the `border` using the `--border-default` CSS variable, and set the `border-radius`. Override browser defaults for `font-size` and `outline`. Disable `outline` because the border effectively does the same thing. Set `padding-left` to satisfy the mockup and `box-sizing` to `border-box` to make styling more predictable.

```
input {
  height: var(--input-min-dimension);
  width: 100%;
  border-radius: var(--radius-sm);
  border: var(--border-default);
  font-size: var(--font-body-md);
  padding-left: var(--padding-sm);
  outline: none;
  box-sizing: border-box;
}
```



Since you've disabled the outline that would normally appear when the input is focused, return the user feedback by setting the `border` to a different color using the CSS pseudo selector `:focus`, making sure to also override `:focus:hover` and `:active`. This styling will make the border darker whenever the input is focused or active.

```
input:focus,
input:focus:hover,
input:active {
  border: var(--border-focus);
}
```



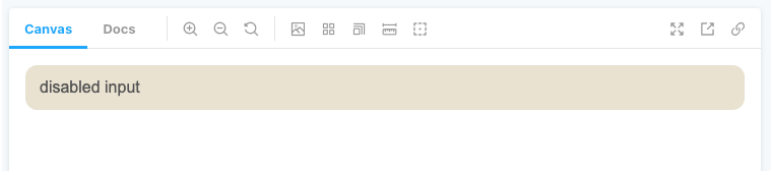
Style the Error story

Navigate to the `Error` story in Storybook to begin styling the error state. Select the `.error` class with CSS while also targeting all the pseudo classes so that whenever the form control has an error state, the border and font color changes to bright red.

```
    input.error,
    input.error:hover,
    input.error:focus,
    input.error:active {
      border: var(--border-error);
      outline: none;
      box-shadow: none;
      color: var(--color-error);
    }
```

The element that contains the error messages also requires some styling. Select the `.message` class with CSS and apply the styles from the mockup.

```
    .message {
      margin-top: var(--margin-xxs);
      color: var(--color-error);
      font-weight: var(--font-weight-default);
    }
```



Style the Disabled story

After you've observed the new error styling taking effect in Storybook, navigate to the `Disabled` story to style the disabled state. To select the disabled state with CSS, target the attribute using square brackets. Change the `opacity`, `background`, and `border` styles using the appropriate CSS variables, making sure to also override the `:focus`, `:hover`, and `:active` states.

```
    input[disabled] {
      opacity: var(---color-disable);
      background: var(--color-disable);
      border: var(--border-disable);
    }
    input[disabled]:hover,
    input[disabled]:focus,
    input[disabled]:active {
      border: var(--border-disable);
      outline: none;
      box-shadow: none;
    }
```

Styling `TextInputComponent` in the context of the template means the styling is encapsulated by Shadow DOM. This is one of the clear benefits of using Shadow DOM, although as we've seen, this encapsulation makes it impossible for the `HTMLInputElement` to fully participate in `HTMLFormElement` without also using the APIs provided by a form-associated custom element. In the next sections, you'll wrap up development of `TextInputComponent` by granting the component all the same features of the `HTMLInputElement` and making sure the `HTMLFormElement` state is updated whenever the user changes the value of the form control.

Emulating HTMLInputElement

In previous sections, you declared getters and setters for `required`, `disabled`, and `value` and used `observedAttributes` to watch for changes to these attributes on `TextInputComponent`, but what about the rest of the attributes usually provided by `HTMLInputElement` that are still missing on `TextInputComponent`? Since the input element is encapsulated by Shadow DOM, it's not easily accessible from the outside world. Developers could access the input element using

the `$input` property, but things could get out of sync between `TextInputComponent` and `HTMLInputElement` fairly quickly.

It's a fairly safe bet to assume that state is managed by `HTMLInputElement` because it does a lot of heavy lifting, providing attributes and properties that handle fairly complex problems. Setting `pattern` on `HTMLInputElement`, for instance, allows the developer to ensure the value of the input matches `RegExp`, while `list` allows developers to specify the `id` of a `<datalist>` element that contains autocomplete options. Without `HTMLInputElement`, you'd have to figure out how to implement some of these complex features. When coding form-associated custom elements, it makes a lot of sense to reuse behaviors of existing inputs, even if it means the input is completely hidden from the view, like you will see in later chapters when you develop `CheckBoxComponent`.

Ensure that `TextInputComponent` has many of the same features of `HTMLInputElement` by implementing the same attributes, getters, and setters that are also provided by an input element where type is equal to `text`.

Update `observedAttributes` to listen for changes on several attributes usually found on `HTMLInputElement`.

```
1   static get observedAttributes() {
2     return [
3       "name",
4       "type",
5       "required",
6       "minlength",
7       "maxlength",
8       "pattern",
9       "list",
10      "placeholder",
11      "readonly",
12      "spellcheck",
13      "disabled",
```

```
      "value",
    ];
  }
```

Add cases for each of the attributes in `attributeChangedCallback`, setting the appropriate attribute of the `HTMLInputElement` whenever the attribute on `TextInputComponent` is updated.

```
  attributeChangedCallback(name, prev, next) {
    this.$attr[name] = next;
    switch (name) {
      case "value":
        this.value = next;
        break;
      case "disabled":
        this.disabled = next;
        break;
      case "required":
        this.required = next;
        break;
      case "type":
        this.$input.setAttribute("type", next);
        break;
      case "minlength":
        this.$input.setAttribute("minlength", next);
        break;
      case "maxlength":
        this.$input.setAttribute("maxlength", next);
        break;
      case "pattern":
        this.$input.setAttribute("pattern", next);
        break;
      case "list":
        this.$input.setAttribute("list", next);
        break;
```

```
      case "placeholder":
        this.$input.setAttribute("placeholder", next);
        break;
      case "readonly":
        this.$input.setAttribute("readonly", next);
        break;
      case "spellcheck":
        this.$input.setAttribute("spellcheck", next);
        break;
    }
  }
```

type

Code getters and setters for `type`. The getter should return `text` by default or `type` when it's present on `this.$input`. Set the type using `this.$input.setAttribute`.

```
  get type() {
    return this.$input.type ?? "text";
  }
```

```
  set type(type: string) {
    this.$input.setAttribute("type", type);
  }
```

list

`list` is only a getter on `HTMLInputElement`. The only way to set `list` is via `setAttribute`, which is handled in `attributeChangedCallback`.

```
  get list() {
    return this.$input.list;
  }
```

Set up getters and setters for `minLength`, which establishes the minimum length of characters for the value of the input.

```
  get minLength() {
    return this.$input.minLength;
  }
```

```
  set minLength(min: number) {
    this.$input.minLength = min;
  }
```

maxLength

Code getters and setters for `maxLength`, which establishes the maximum number of characters for the value of the input.

```
  get maxLength() {
    return this.$input.maxLength;
  }
```

```
  set maxLength(max: number) {
    this.$input.maxLength = max;
  }
```

readOnly

`readOnly` is only a getter on `HTMLInputElement`, so replicate the same readonly property on `TextInputComponent`.

```
  get readOnly() {
    return this.$input.readOnly;
  }
```

pattern

`pattern` can be set via `setAttribute` or a setter on `HTMLInputElement`, so replicate the same getters and setters on `TextInputComponent`.

```
  get pattern() {
    return this.$input.pattern;
  }
```

```
  set pattern(pattern: string) {
    this.$input.pattern = pattern;
  }
```

placeholder

`placeholder` is typically set on `HTMLInputElement` with `setAttribute`, but the element also provides a getter.

```
  get placeholder() {
    return this.$input.placeholder;
  }
```

spellcheck

`spellcheck` is typically set on `HTMLInputElement` with `setAttribute`, but the element also provides a getter.

```
  get spellcheck() {
    return this.$input.spellcheck;
  }
```

All those getters and setters may seem like a lot of busywork, but they go a long way to providing a stable API for developers. When developers implement user interfaces with `TextInputComponent`, they can expect the same API as `HTMLInputElement` directly on the form-associated custom element. For the sake of brevity, we've omitted some APIs involved with making selections in `HTMLInputElement`, but even those could be added to the `TextInputComponent` using the same techniques. For a complete reference, read The Input (Form Input) element on MDN[46].

Making a form

In the final push to implement the rest of `TextInputComponent`, you'll need another story to test how the form control participates in form submission. This story will support two inputs for a login view: one input for username and another that handles the password. This will allow you to also test some of the new features added in the last section, namely `type`, `minlength`, and `pattern`.

The login form must be validated in such a way that both inputs are required and have a minimum length, while the password input should include at least one lowercase letter, uppercase letter, number, and special character.

Update the `validators Object` in TextInput.stories.js. Add another property called `password`. Give both `username` and `password` the following validations:

[46] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#input_types

```
    validations: [
      {
        flag: { valueMissing: true },
        message: "Error: Required, please enter a usernam\
e.",
        condition: (input) => input.required && input.val\
ue.length <= 0,
      },
      {
        flag: { tooShort: true },
        message:
          "Error: Minimum length not met, please supply a\
 value with at least 8 characters.",
        condition: (input) =>
          input.minLength && input.value.length < input.m\
inLength,
      },
    ],
```

Add another validation to the password validator that validates against the `pattern`.

```
      {
        flag: { patternMismatch: true },
        message:
          "Please use at least one uppercase, uppercase l\
etter, special character, and number.",
        condition: (input) =>
          input.pattern &&
          input.value.match(new RegExp(input.pattern)) ==\
= null,
      },
```

Declare a new template called `FormTemplate`. This template injects a few more arguments than previously: `headline`, `onSubmit`, `onFormData`, in addition to `onValidate`.

```
const FormTemplate = ({ headline, onSubmit, onValidate, o\
nFormData }) => {
```

Just like the other templates, this `function` returns a call to `html` followed by a template string that includes the HTML template. Place a new h4 element before the form element and bind the `headline` argument to its content. Bind the two new callbacks `onFormData` and `onSubmit` the same way as `onValidate`, using the `@` template syntax provided by LitHTML that calls `addEventListener` on the form element.

```
  return html` <h4 slot="header">${headline}</h4>
    <form
      name="foo"
      slot="content"
      @formdata="${onFormData}"
      @validate="${onValidate}"
      @submit="${onSubmit}"
    >
```

Add a fieldset to the form. Place a label with the name of the input prior to each `in-textinput`, ensuring the label has a `for` attribute that matches the `name` attribute of the in-textinput it describes. We'll go into much better detail about this relationship in the next chapter when discussing accessibility.

```
      <fieldset>
        <legend>Login Form</legend>
        <label for="username">Username</label>
        <in-textinput
          type="text"
          id="username"
          name="username"
          required
          minlength="8"
          class="form-control"
        ></in-textinput>
        <label for="password">Password</label>
        <in-textinput
          type="password"
          id="password"
          name="password"
          required
          minlength="8"
          pattern="^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(\
?=.*?[#?!@$ %^&*-]).{8,}$"
          class="form-control"
        ></in-textinput>
```

Make sure each in-textinput has the appropriate `type` attribute, is marked `required`, and has the `minlength` set to `8`. The password input should have an additional `pattern` attribute with the provided `RegExp` that validates the password meets the minimum requirements.

```
^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$ %^&*-])\
.{8,}$
```

Don't forget to add the `$validator` property to each input so that both form controls take advantage of the `validate` function you coded earlier.

```
  setTimeout(() => {
    for (let prop in validators) {
      document.querySelector(`[name="${prop}"]`).$validat\
or = validators[prop];
    }
  }, 0);
```

Add another input to the template at the end of the `fieldset`, this time with the `type` attribute set to `submit`.

```
        <input class="submit" type="submit" value="Submit\
" />
```

Inputs of the type `submit` are how browsers handle form submission by default. When the input is rendered as a button, clicking on this element will make the browser user agent attempt form submission. Single-page applications may not use that approach, but in the story, the input will provide a handy means to test `TextInputComponent` is participating in form submission.

When you are finished, `FormTemplate` should look like this:

```
const FormTemplate = ({ headline, onSubmit, onValidate, o\
nFormData }) => {
  setTimeout(() => {
    for (let prop in validators) {
      document.querySelector(`[name="${prop}"]`).$validat\
or = validators[prop];
    }
  }, 0);

  return html` <h4 slot="header">${headline}</h4>
    <form
      name="foo"
      slot="content"
```

```
14        @formdata="${onFormData}"
15        @validate="${onValidate}"
16        @submit="${onSubmit}"
17      >
18        <fieldset>
19          <legend>Login Form</legend>
20          <label for="username">Username</label>
21          <in-textinput
22            type="text"
23            id="username"
24            name="username"
25            required
26            minlength="8"
27            class="form-control"
28          ></in-textinput>
29          <label for="password">Password</label>
30          <in-textinput
31            type="password"
32            id="password"
33            name="password"
34            required
35            minlength="8"
36            pattern="^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(\
37  ?=.*?[#?!@$ %^&*-]).{8,}$"
38            class="form-control"
39          ></in-textinput>
40          <input class="submit" type="submit" value="Submit\
41  " />
42        </fieldset>
43      </form>`;
44  };
```

Export a new const called `Form` and call `bind` on the `FormTemplate function`.

```
export const Form = FormTemplate.bind({});
```

Declare several new properties on the story `args` for the template. These properties get passed to the `FormTemplate` function you just declared. First, set the `headline` property to `Login` so that the proper headline appears in the h4.

```
Form.args = {
  headline: "Login",
```

When you click on the submit button, the template is expecting a callback named `onSubmit`. In this callback, you'll want to call `preventDefault` on the event. Otherwise, Storybook will try to navigate away when you click on the button. In this callback, create a new `FormData Object`, passing in the `HTMLFormElement` into the first argument. Wrapping that in `console.log` will allow you to inspect the `FormData`.

```
  onSubmit: (ev) => {
    console.log(new FormData(ev.target));
    ev.preventDefault();
  },
```

Oddly enough, just by instantiating the `FormData` in this callback, you've triggered another event on the form element called `formdata`, which you've already bound to the element in the `FormTemplate` with the callback `onFormData`. Declare `onFormData` on the `args Object`, being sure to log the event.

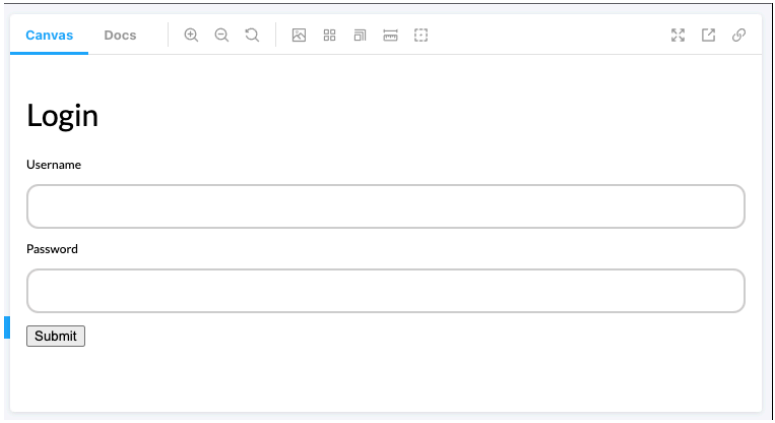
```
  onFormData: (ev) => {
    console.log(ev);
    for (let value of ev.formData.values()) {
      console.log(value);
    }
  },
```

One way to inspect the values of each input is to loop over the `values` property on the `formData Object`, as seen in the above example. The `value` logged there is the value of each input on form submission.

Add one more property to the `args` for validation. Similar to the callback in the `Primary` story, `onValidate` checks the truthiness of the `validity.valid` on each input, but this time in a loop because you're dealing with more than one input.

```
  onValidate: (ev) => {
    const validations = [];
    for (let prop in validators) {
      validations.push(
        document.querySelector(`[name="${prop}"]`).validi\
ty.valid
      );
    }
    if (validations.filter((val) => val == false).length)\
 {
      console.warn("INVALID");
    } else {
      console.log("VALID");
    }
  },
```

When you are finished, the `Form` story should appear like below.



That's it for story creation! Now let's wire up the final parts of `TextInputComponent` so the form control can participate in form submission.

Handling FormData

If you click the submit button with both inputs passing validation and then inspect the `FormData Object` in Dev Tools, you'll notice the `entries` and `values` Iterators are empty. That means `TextInputComponent` still isn't fully working with form submission. This is actually fairly easy to fix, but it means exploring two more lifecycle hooks provided by form-associated custom elements, `formStateRestoreCallback` and `formResetCallback`, which handle form state at different times. You'll also use another method on `ElementInternals` to update the form with the value of the input: `setFormValue`.

To update the form with the value of the input, we should consider when the user updates the value of the input. As of yet, you've only handled changes to the input value on blur in the `connectedCallback`, but you'll need to also attach a callback, listening to the `change` event on the `HTMLInputElement`.

In `TextInput.ts`, add the following to the `connectedCallback`:

```
    this.$input.onchange = () => {
      this.onChange();
    };
```

Declare a method on `TextInputComponent` called `onChange`. In this method, call `this.internals.setFormValue`, passing in the value of the input into both arguments. The first argument of `setFormValue` sets the value of the input on the `FormData Object`, which fulfills the requirement during form submission.

```
  onChange() {
    this.shadowRoot.querySelector(".message").innerHTML =\
 "";
    this.$input.classList.remove("error");
    this.internals.setFormValue(this.value, this.value);
  }
```

While you're in this method, clean up some of the logic from validation. Clear out the messages in the div by setting `innerHTML` to an empty string, and remove the `error` class from the `HTMLInputElement`. This will have the effect of clearing the error state when the user types.

The second argument of `setFormValue` is for when the user navigates away from the form and back again. The second argument allows the browser to retain the value of the input element so the user doesn't have to enter the value again, but only when the `formStateRestoreCallback` is defined.

```
  formStateRestoreCallback(state: string, mode: string) {
    this.value = state;
  }
```

`formStateRestoreCallback` is called when the browser wants to restore the state of form control. Use-cases may include when the user navigates away from the form and back again, the browser reloads, or when the browser wants to autofill the value. The second argument of `setFormValue` is optional, but we'll use it and the `formStateRestoreCallback` for the time being. However, with further development, setting this argument could be configurable via the `TextInputComponent` interface.

Similarly, `formResetCallback` is called whenever the form resets, and each form control could return to its original value. This callback is a convenient means to reset the value of the form control to whatever is specified by the `value` attribute.

```
1   formResetCallback(state: string) {
2     this.value = this.getAttribute("value") || "";
3   }
```

In Storybook, enter values into both inputs that pass validation and click the **Submit** button. Observe the `FormData` now has values for each input thanks to `setFormValue`.

Testing TextInputComponent

Integration tests are important for `TextInputComponent` because you can validate the component's outward facing API: the attributes, getters, and setters you defined in previous sections operate as expected. Snapshot tests will validate the interaction states you styled.

For the integration tests, you'll be using the `Form` Story from the previous section as a test bed. Before you begin writing integration tests, ensure there is a `submit` classname added to the submit button in the `FormTemplate` so it can be easily queried during testing.

```
        <input class="submit" type="submit" value="Submit\
" />
```

Make a new file in the cypress/integration directory named `TextInput.spec.js`.

```
touch cypress/integration/TextInput.spec.js
```

Ensure Storybook is running in a Terminal window.

```
yarn storybook
```

In a separate Terminal tab or window, start up Cypress in open mode.

```
yarn test:e2e:open
```

In `TestInput.spec.js`, set up the test by declaring a `describe` block.

```
describe("TextInputComponent", () => {

}
```

Make an assertion that tests if the error message displays on blur. First tell Cypress to navigate to the Form story in Storybook, referencing the URL of the story in the browser address bar to set a query param for `iframe.html`.

```
  it("should display error message on blur", () => {
    cy.visit("iframe.html?id=components-inputs-textinput-\
-form");
```

Query for each element in Light DOM with `get`, checking each element is visible using `should`, and then use Cypress to click on each input and the submit button by chaining `click()`.

```
    cy.get("#root").get('[id="username"]').should("be.vis\
ible").click();
    cy.get("#root").get('[id="password"]').should("be.vis\
ible").click();
    cy.get("#root").get(".submit").should("be.visible").c\
lick();
```

Find the message container elements in Shadow DOM, and check they contain the appropriate error messages using similar chained methods, but this time querying in ShadowDOM using the `shadow` and `find` methods.

```
    cy.get("#root")
      .get('[id="username"]')
      .shadow()
      .find(".message")
      .contains("Error: Required, please enter a username\
.");
    cy.get("#root")
      .get('[id="password"]')
      .shadow()
      .find(".message")
      .contains(
        "Minimum length not met, please supply a value wi\
th at least 8 characters."
      );
```

Test that both inputs in the `FormTemplate` don't display error messages when the inputs are valid. Write a new test titled `should not display error message when valid` and visit the same story in Storybook.

```
  it("should not display error message when valid", () =>\
 {
    cy.visit("iframe.html?id=components-inputs-textinput-\
-form");
```

Select each input, and call `type` to enter a value that should pass validation.

```
    cy.get("#root")
      .get('[id="username"]')
      .shadow()
      .find("input")
      .type("jane@doe.com");
    cy.get("#root")
      .get('[id="password"]')
      .shadow()
      .find("input")
      .type("W3BC0mpon3nts!");
```

Use Cypress to click the submit button.

```
    cy.get("#root").get(".submit").should("be.visible").c\
lick();
```

Verify that both message containers are empty.

```
    cy.get("#root")
      .get('[id="username"]')
      .shadow()
      .find(".message")
      .should("include.text", "");
    cy.get("#root")
      .get('[id="password"]')
      .shadow()
      .find(".message")
      .should("include.text", "");
```

You just tested the validations you developed in this chapter work when the value of `TextInputComponent` is invalid and that no user feedback is visible when `TextInputComponent` is valid. There are many more aspects to the `TextInputComponent` that could be validated with integration tests, although we'll stop here for the sake of brevity.

Make sure you stop the process running Cypress before proceeding with the snapshot tests.

Snapshots

With Storybook still running generate snapshots for the component.

```
yarn test:snapshot:ci
```

After snapshots have been generated for each TextInput story, add them to your commit. Commit your changes in git to complete this section. Now you've completed integration and snapshot tests for `TextInputComponent`. Let's review the content of Chapter Two.

Summary

In this chapter, you accomplished a lot! You developed a form-associated custom element, `TextInputComponent`, that handles several states for a text input: default, error, and disabled. By setting `formAssociated` on the component `class`, you enabled `TextInputComponent` to participate in form validation and submission. `ElementInternals` provided several methods to aid in validation, but didn't provide a handy method for validating an entire form using the value from `TextInputComponent`, so you dispatched a `validate Event` to the `HTMLFormElement`, which allowed you to hook into a custom validation model. `setFormValue` on `ElementInternals` allowed you to update form `FormData` and loop over the form's entries on submission.

Later in this book in Chapter Nine, you'll reuse some of the functionality mocked in Storybook to develop the login view in the application.

Examples in this chapter are practical, so the potential of form-associated custom elements may not be obvious. Adding `ElementInternals` to a custom element gives you the best of both worlds:

- Template and styling encapsulation with Shadow DOM
- The ability to participate in form validation and submission

Form-associated custom elements don't have to include a typical form control like `input` or `textarea`. You can craft new user interfaces and with `ElementInternals` have these custom experiences participate in forms like any other form control. That is the real potential of form-associated custom elements: completely new user interfaces that work seamlessly with HTML forms as if they were native form control in the browser.

In the next chapter, you'll expand on the topic of forms by making the `TextInputComponent` follow Accessibility best practices, and

you'll enhance the submit button with another Web Components specification: customized built-in elements.

Customized built-in elements

Chapter Three

Introduction

The internet should be a place that is accessible and inclusive for everyone. When coding for the web browser, developers should consider users that require assistive devices to interact with the content. It could even be required by law. To conduct business with the U.S. federal government or follow best practices in accordance with the European Trade Commission (EU), web developers need to follow Web Content Accessibility Guidelines[47] defined by the W3C Web Content Accessibility Group (WCAG).

In the previous chapter, I mentioned that `ElementInternals` provides the entire Accessibility Object Model (AOM)[48] to a form-associated custom element. AOM is a browser specification that attempts to reduce the friction developers face when implementing accessible user interfaces. AOM provides properties on the DOM node that allow developers to programmatically read and write accessibility-based properties.

Prior to the addition of AOM, developers were limited to WAI-ARIA, a set of HTML attributes that provide context to assistive technologies. By providing an attribute such as `aria-labelled-by`, WAI-ARIA allowed engineers to provide context to screen readers.

[47] https://www.w3.org/TR/WCAG20/
[48] https://wicg.github.io/aom/spec/

```
<div id="buttonLabel">Do you agree?</div>
<input type="checkbox" aria-labelled-by="buttonLabel" />
<div role="button">Continue</div>
```

Web developers tend to choose the path of least resistance, whether pressured by product owners or lack of knowledge. As an example, let's look at the `button type="submit"` from the last chapter. Developers may not choose `button` element because the default styling is cumbersome to override. For whatever reason, a developer may opt to use a `div` and apply the WAI-ARIA `role` attribute to notify assistive devices this `div` is actually a `button`.

```
<div role="button">Submit</div>
```

If you used the techniques in the previous two chapters to develop a new button component using autonomous custom elements, you'd declare a new `class` that extends from `HTMLElement`, although this comes at a similar cost as the example of the `div`. Since you're not implementing a `HTMLButtonElement`, you lose all the accessibility characteristics of a native button, namely the ability for users to press the button using only the keyboard. `HTMLButtonElement` allows users to press Enter or Spacebar on their keyboard instead of a mouse click. To gain these same accessible behaviors, you'd have to implement them on the `class` that extends `HTMLElement`.

Customized built-in elements[49] allow you to inherit the characteristics of native browser elements by giving you the ability to extend from their base `class`. By extending from `HTMLButtonElement` instead of `HTMLElement`, the component inherits all the behaviors of the native button element.

```
const ButtonComponent extends HTMLButtonElement
```

[49] https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-customized-builtin-example

Registering customized built-in elements is done the same way. However, calling `define` requires a third argument that specifies an `Object` that sets the `extends` property to the tag name of the element that you're extending from.

```
customElements.define("in-button", ButtonComponent, { ext\
ends: "button" });
```

To use the customized built-in element in a template, the element must use the required `is` attribute set to the name of the element in the `CustomElementRegistry`. This is mainly so that browsers can parse the custom element from HTML source.

```
<button is="in-button">Submit</button>
```

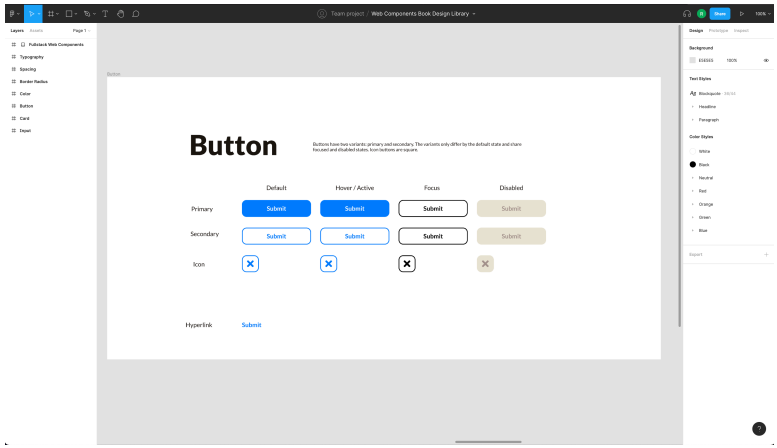
Customized built-in elements can be created dynamically with `document.createElement` by adding a second argument when calling the function.

```
const button = document.createElement("button", { is: "in\
-button" });
```

At first, it may seem a little awkward that customized built-ins require different syntactical sugar or may behave differently from autonomous or form-associated custom elements, but the use value of customized built-in elements is undeniable. Being able to inherit user-agent supplied behaviors when developing custom elements ensures users are already familiar with the characteristics of the component. The complexity of the component source is reduced when developers don't have to re-implement user-agent based behaviors. Customized built-in elements are essential in terms of accessibility because the component retains accessible behaviors supplied by the browser.

What you will build

One obvious use case for a customized built-in element is a button. In this chapter, you'll code `ButtonComponent`, a customized built-in element that is used throughout the application. The component must work in and out of the context of `HTMLFormElement`. Looking at Figma[50], `ButtonComponent` has three different variants: primary, secondary, and icon. The component has different states for hover, active, focus, and disabled.



This exercise mostly involves styling the different states of `ButtonComponent`. Since you'll retain all the accessibility characteristics of a button, there's no need to include any logic on the `class` other than dealing with the various interaction states. If engineers wish to customize the button further, they can do so by extending the `ButtonComponent class`.

Only some native elements can encapsulate styling and template with Shadow DOM: `HTMLBodyElement`, `HTMLDivElement`, `HTMLHeadingElement`, `HTMLParagraphElement`, and `HTMLSpanElement`,

[50] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=317%3A217

which means the CSS for most customized built-in elements has to be in a `<style>` tag external to the element. You could, as a rule of thumb, add CSS classnames to customized built-in elements so they could be targeted via external stylesheet. Traditionally, the CSS would be declared in the `<head>`, but if the customized built-in element is used within the context of another element's `ShadowRoot`, that styling won't be able to cross the shadow boundary, leaving the element unstyled. To work around this limitation, later in this book, you're going to code a `function` that traverses DOM for the nearest `ShadowRoot` and injects the styles there. Since the button is slotted into the `CardComponent` template, it's exposed to Light DOM and can be styled with a stylesheet in the `document.head`. As you code more complex custom elements, this will not always be the case.

After completing the styling for `ButtonComponent`, you'll circle back and fix accessibility issues that remain in the form. Using a screen reader, you'll make the form accessible by utilizing the Accessibility Object Model on the `ElementInternals` interface or through the use of WAI-ARIA attributes.

What you will learn

- What is a customized built-in element?
- What are the benefits of a customized built-in element?
- What is the Accessibility Object Model? (AOM)
- How do I provide a label for screen readers?
- How do I use a screen reader to test for accessibility?
- How do I provide an accessible validation message for screen readers?
- Why doesn't a customized built-in element work in Safari?

Acceptance criteria

- `ButtonComponent` displays primary, secondary, and icon buttons
- `ButtonComponent` is styled properly in the context of another element's `ShadowRoot`, or via stylesheet declared in `<head>`
- User can navigate the form using a screen reader

Getting started

Navigate to the project directory.

```
cd fullstack-web-components
```

Checkout a new branch named chapter-3 from where you left off at the end of chapter 2.

```
git checkout -b chapter-3
```

Run Storybook using `yarn`. Storybook should appear, and the development environment is ready.

```
yarn storybook
```

Scaffolding ButtonComponent

If the following steps look familiar by now, that's because they resemble the same process for scaffolding `TextInputComponent` and `CardComponent` from the previous two chapters. First, you'll declare a new custom element, this time registering the component as a

customized built-in element. Then, you'll import that component into a new story file where all the `ButtonComponent` stories will be declared. Throughout the chapter, you will code several stories to visualize the different variants of the `ButtonComponent`: primary, secondary, and icon, as well as at least one story that demonstrates the disabled state.

To get started, make a new TypeScript file called `Button.ts` in a new directory: `packages/component/src/button`.

```
mkdir -p packages/component/src/button
```

In the input directory, make a new file and name it `TextInput.ts`.

```
touch packages/component/src/button/Button.ts
```

In `Button.ts`, export a new `class` named `ButtonComponent` that extends `HTMLButtonElement`. Give the `class` a `constructor`, making sure to call `super` so that when the component initializes, it also instantiates the constructor of the `HTMLButtonElement class`.

```
export class ButtonComponent extends HTMLButtonElement {
  constructor() {
    super();
  }
}
```

One difference you should notice by now between `ButtonComponent` and the last two components you coded is the `class` no longer extends from `HTMLElement` but instead extends from `HTMLButtonElement`, an interface that provides methods and properties common to the native `button` element. Find a complete reference of `HTMLButtonElement` on MDN[51].

[51] https://developer.mozilla.org/en-US/docs/Web/API/HTMLButtonElement

There are at least two advantages gained by extending from `HTMLButtomElement` as opposed to `HTMLElement`. `HTMLButtonElement` natively participates in forms by implementing the `form`, `formAction`, `formEnctype`, `formMethod`, `formNoValidate`, and `formTarget` properties. If you had opted for a form-associated custom element, you would have to provide the `ElementInternals` interface to make the button participate in the form. In the above example, you don't have to do anything. You get form participation for free. By extending from a native element, the component inherits all the behaviors of that element. Another example may not be evident looking at the documentation for `HTMLButtonElement`. The button element includes features that promote accessibility out of the box, including the ability to press the button using the Enter key or Spacebar. Had you chosen to extend from `HTMLElement`, you would have to duplicate this functionality with a custom implementation.

Like the previous two components, you register the `ButtonComponent` on the `CustomElementRegistry`, this time passing in the optional third argument, an `Object` with `extends` property set to `button`. The `extends` property is necessary to tell the browser you're registering the element that progressively enhances a native element.

```
1  customElements.define("in-button", ButtonComponent, { ext\
2  ends: "button" });
```

Create a new file named `Button.stories.js` in the same directory as `Button.ts`.

```
1  touch packages/component/src/button/Button.stories.js
```

Import `ButtonComponent` into `Button.stories.js`.

```
import { ButtonComponent } from "./Button";
```

Declare the story by exporting a default `Object` that sets the `title` property as `Components/Inputs/Button` and the `component` property as `in-button`.

```
export default {
  title: "Components/Inputs/Button",
  component: "in-button",
};
```

Before you declare the first story template, import `html` from the lit-html package since Storybook uses LitHTML to render Web Components.

```
import { html } from "lit-html";
```

Declare a new `const` named `Template`, and make it equal to a `function` that returns `html`. Pass a new template string to `html`, and include a button element in the template, making sure to use start and end button tags. A quirky requirement of customized built-in elements is the requirement of declaring the `is` attribute on the button element, setting the name of the registered element, in this case, `in-button`.

```
const Template = ({ label, variant }) => html` <button
  class="${variant}"
  is="in-button"
>
  ${label}
</button>`;
```

To support the three button variants: primary, secondary, and icon, you could set an attribute and use `observedAttributes` and

`attributeChangedCallback`, but that seems like overkill when you could achieve the same with a CSS classname. The variants, after all, are just various styles of the same button element. We need a way to set the variant in the story, so pass in the `variant` using destructuring to the `class` attribute. Finally, pass in a `label` in the same way, setting the content of the button element with the value of `label`.

Next, export a `const` named `Primary` and make it equal to `Template.bind({})`. Set the `Primary.args`, passing in the `variant` and `label`. This establishes the first story for the primary variant in Storybook.

```
export const Primary = Template.bind({});
Primary.args = {
  variant: "primary",
  label: "Button",
};
```

Styling ButtonComponent

While extending `HTMLButtomElement` has benefits, it means that Shadow DOM is not available in the context of `ButtonComponent`. That's because the native button element cannot have a `ShadowRoot`, so we lose the ability to encapsulate styling for the customized built-in element. That means you'll have to rely on CSS styles in a `<style>` tag external to the button element. Since you won't be able to target the button with `:host`, specify a unique CSS class name that can be used to select the custom element. The only place to do this safely in a customized built-in element is the `connectedCallback` lifecycle.

In `Button.ts`, add the `connectedCallback` method and use the `add` method on `classList` to give the button a class name when added to DOM.

```
  connectedCallback() {
    this.classList.add("in-button");
  }
```

Inspect the element with Dev Tools and notice the `primary` and `in-button` class names are set, the first by Storybook and the second through the `connectedCallback`. The element should have the `is` attribute set to `in-button`, the name of the registered customized built-in element.



Primary

You don't have to give up modern conveniences like CSS-in-JS even though the CSS has to be declared in an external `<style>` tag that can be accessed by instances of `ButtonComponent`. In `Button.ts`, declare a `const` named `buttonStyles` and make it equal to a template string. Select the primary button variant with the selector `.in-button.primary` and set the styling per the mockup in Figma.

```
const buttonStyles = `
  .in-button.primary {
    background: var(--color-blue-500);
    border: 2px solid var(--color-blue-500);
    box-sizing: border-box;
    border-radius: 12px;
    min-height: 44px;
    min-width: 180px;
    color: var(--color-white);
    font-size: var(--font-body-md);
    font-weight: var(--font-weight-button);
    cursor: pointer;
    padding: 0px;
  }
`;
```

To get the button styles in DOM, add a new method on the `ButtonComponent class` called `addStylsheet`.

```
  addStylesheet() {
    const head = document.head;
    if (document.getElementById(`in-button-style`)) {
      return;
    }
    const style = document.createElement('style');
    style.setAttribute('id', `in-button-style`);
    style.textContent = buttonStyles;
    head.appendChild(style);
  }
```

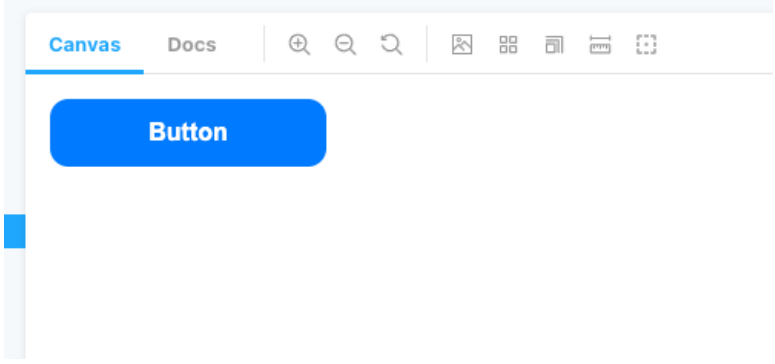
In this method, create a new style element using `document.createElement` and set the `id` attribute to `in-button-style`. Using `textContent`, set the `buttonStyles` you declared earlier as the content of the style element, then append the style element to the `document.head`. When there are

multiple instances of `ButtonComponent` in the same view, we don't want to keep adding the same stylesheet to `document.head`, so implement a conditional that prevents this from happening.

When you are finished coding `addStylesheet`, make sure to call the method in the `connectedCallback`.

```
1   connectedCallback() {
2     this.classList.add('in-button');
3     this.addStylesheet();
4   }
```

In Storybook, you should see a button that is styled as the primary variant.



Inspect the `<head>` tag in Dev Tools and observe a new `<style>` tag with the CSS declared as `buttonStyles` injected into the style element. If you have trouble seeing the tag, ensure the JavaScript context is set to storybook-preview-iframe in Dev Tools.

```
<style>…</style>
<style id="in-button-style"> == $0
    .in-button.primary {
      background: var(--color-blue-500);
      border: 2px solid var(--color-blue-500);
      box-sizing: border-box;
      border-radius: 12px;
      min-height: 44px;
      min-width: 180px;
      color: var(--color-white);
      font-size: var(--font-body-md);
      font-weight: var(--font-weight-button);
      cursor: pointer;
      padding: 0px;
    }
  </style>
</head>
```

That's it for the default primary styles. Style the secondary and icon variants before you work on the styles for each button state: disabled, active, and focused. That'll save time because looking at the Figma, it appears all the buttons share the styling for the various states, while they differ in their default styles.

Secondary

Declare another story in `Button.stories.js` for the secondary button variant, naming the story `Secondary` and making sure to call `Template.bind({})`. Pass in `variant` and `label` through `Secondary.args`, setting the `variant` to `secondary`.

```
export const Secondary = Template.bind({});
Secondary.args = {
  variant: "secondary",
  label: "Button",
};
```

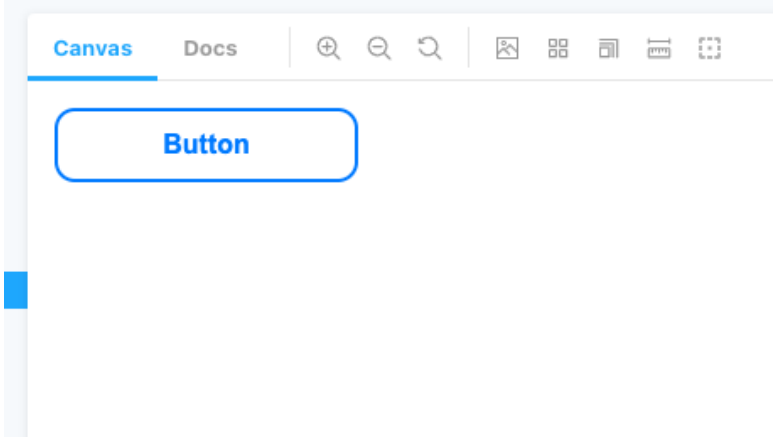
In Figma, the secondary buttons appear to have a white background with a blue border. The label is the same color as the border. Update

`buttonStyles` with the CSS for the secondary button. To select secondary buttons, use the `.in-button.secondary` selector.

In `Button.ts`, style the element based on the mockup in Figma by adding the secondary styles to `buttonStyles`.

```
.in-button.secondary {
  background: var(--color-white);
  border: 2px solid var(--color-blue-500);
  box-sizing: border-box;
  border-radius: 12px;
  min-height: 44px;
  min-width: 180px;
  color: var(--color-blue-500);
  font-size: var(--font-body-md);
  font-weight: var(--font-weight-button);
  cursor: pointer;
  padding: 0px;
}
```

When you are finished, the default secondary button should be styled like the example below.



Icon

Scaffolding the icon button variant requires a little more work than primary and secondary. You'll need a way to source an SVG to place in the button in lieu of the text label. FontAwesome is installed in the repository to help you achieve this goal. In Figma, the example icon button is used throughout the application as the "close" button. This icon could be used to close dialogs and menus. The closest icon FontAwesome provides is a "plus" icon, so you'll need to use CSS to rotate the "plus" 45 degrees to display a "cross." You can achieve this style through a CSS classname modifier specific to the close button. While all icon variants should use the CSS classname `icon`, this specific icon will require the CSS classname `icon-close`.

FontAwesome provides a package called `@fortawesome/fontawesome-free`, which is already installed in the repository and integrated with Storybook. In `Button.stories.js`, use the `window.FontAwesome.icon` method to select the "plus" SVG. Set a `let` named `icon` to the SVG returned by `window.FontAwesome.icon`.

```
let icon = null;

if (window.FontAwesome) {
  icon = window.FontAwesome.icon({ prefix: "fas", iconNam\
e: "plus" });
}
```

If you inspect `icon`, you'll observe the `Object` provides an interface for consuming an SVG. The SVG can be accessed via `icon.node[0]`. In case you need to process the SVG later, set a new `const` named `svg` to the SVG node.

```
const svg = icon.node[0];
```

Scaffold the Icon story, making sure to export a `const` named `Icon`. This time, specify the `variant` as `icon icon-close`. Since `variant`

merely sets the `class` attribute on the button element in `Template`, the element can now be styled with both classnames. Set the `label` property to the `svg`. The SVG node will be inserted into the content of the button element using LitHTML.

```
export const Icon = Template.bind({});
Icon.args = {
  variant: "icon icon-close",
  label: svg,
};
```

Now that everything is scaffolded for the icon variant, proceed to style the button. In `buttonStyles`, select the icon variant with `.in-button.icon` CSS selector, and while referencing Figma, style the element in the CSS declaration block.

```
  .in-button.icon {
    display: flex;
    align-items: center;
    justify-content: center;
    background: var(--color-white);
    border: 2px solid var(--color-blue-500);
    box-sizing: border-box;
    border-radius: 12px;
    height: 44px;
    width: 44px;
    color: var(--color-blue-500);
    font-size: var(--font-body-icon);
    font-weight: var(--font-weight-button);
    cursor: pointer;
    padding: 0px;
    padding: var(--padding-xs);
  }
```

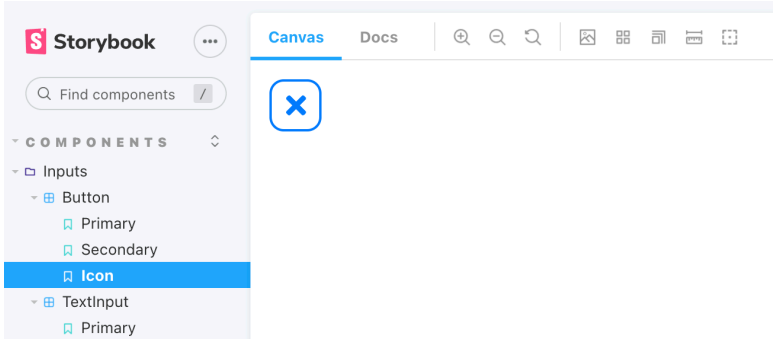
To ensure the SVG fits within the dimensions of the 44px button, select the `svg` with a CSS selector and style its `width` and `height` to

`100%`. Since the icon variant has some padding, the SVG will actually display smaller. In order to transform the "plus" into a "cross", select the `svg` contained by the element with class name `.icon-close` and use CSS `transform` to rotate the SVG `45deg`.

```
.icon svg {
  width: 100%;
  height: 100%;
}

.icon.icon-close svg {
  transform: rotateZ(45deg);
}
```

When you are finished, the icon variant should appear by default as it does in Figma.



Interaction States

Referencing Figma[52] again, you notice each button variant shares the same interaction states. Let's start by styling the focus state. To style the focus state, select each of the variants with the `:focus` CSS

[52] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=0%3A1

pseudo selector. In the CSS declaration block, style the focus state as it appears in Figma.

```
.in-button.primary:focus,
.in-button.secondary:focus,
.in-button.icon:focus {
  background: var(--color-white);
  color: var(--color-black);
  border: 2px solid var(--color-black);
  outline: none;
}
```

The active state is styled in the same way, this time selecting each variant with the `:active` pseudo selector. Style the active state according to the mockup.

```
.in-button.primary:active,
.in-button.secondary:active,
.in-button.icon:active {
  background: var(--color-white);
  color: var(--color-neutral-500);
  border: 2px solid var(--color-neutral-500);
  outline: none;
}
```

All the variants should now have the focus and active states styled. Test out these new behaviors in Storybook by navigating to the `Primary`, `Secondary`, and `Icon` stories. Click on the whitespace in the Storybook canvas and then focus on the button element by pressing the Tab key. With the button focused, press the Spacebar or Return key on the keyboard to see the active state appear.

Buttons have a disabled state as well, usually set with the `disabled` attribute on the button element. To achieve the disabled state in Storybook, declare a new template in `Button.stories.js` named

`DisabledTemplate`. This template should look remarkably similar to `Template`. However, the disabled attribute is set.

```
const DisabledTemplate = ({ label, variant }) => html` <b\
utton
  class="${variant}"
  is="in-button"
  disabled
>
  ${label}
</button>`;

export const Disabled = DisabledTemplate.bind({});
Disabled.args = {
  variant: "primary",
  label: "Submit",
};
```

In `Button.ts`, style the different variants using the CSS attribute selector `[disabled]` and style the disabled state per the mockup in Figma.

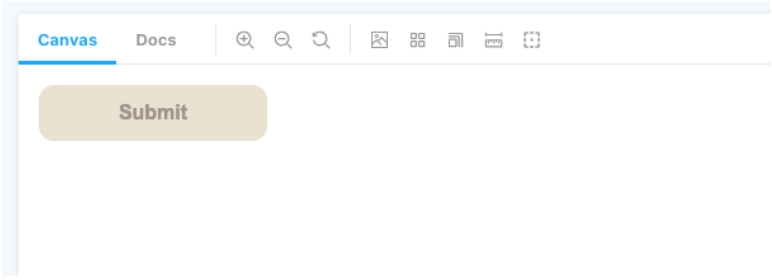
```
.in-button.primary[disabled],
.in-button.secondary[disabled],
.in-button.icon[disabled] {
  opacity: var(---color-disable);
  background: var(--color-disable);
  border: var(--border-disable);
  color: var(--color-neutral-500);
  cursor: default;
}
```

When the button is disabled, the element should not appear focused or active. This means you need a way to override the focus and active styles declared earlier. Using CSS specificity, select each of

the variants with `:focus` and `:active` pseudo selectors, this time increasing the specificity of the selection by adding the `[disabled]` attribute selector. Style the disabled button like the mockup.

```
.in-button.primary[disabled]:focus,
.in-button.secondary[disabled]:focus,
.in-button.icon[disabled]:focus,
.in-button.primary[disabled]:active,
.in-button.secondary[disabled]:active,
.in-button.icon[disabled]:active {
  border: var(--border-disable);
  outline: none;
  box-shadow: none;
}
```

When you are finished styling the disabled state, the button should appear like below.



So far, coding the `ButtonComponent` has been an exercise in styling the different states of the button with CSS. Since the button element cannot use Shadow DOM to encapsulate CSS styles, you injected a stylesheet into the `<head>` and selected CSS classnames applied to the instance of `ButtonComponent` to style the variants of the button. It's worth noting the CSS for `ButtonComponent` could be part of a global stylesheet, but as you'll observe in later chapters, when `ButtonComponent` is a child of another component's `ShadowRoot`, the

shadow boundary will block any global styling. You'll account for this behavior in later chapters.

Since `ButtonComponent` extends from `HTMLButtonElement`, you didn't have to implement behaviors of the button element. If `ButtonComponent` had extended from `HTMLElement` you would have had to implement event listeners to provide the same accessibility behaviors of `HTMLButtonElement`, namely the ability for users to activate the button when pressing the Select or Return keys on the keyboard.

Making the form accessible

The decisions we've made have so far developing the login form resulted in a half-baked accessible experience for users who rely on the keyboard to navigate. `TextInputComponent` has some accessibility limitations. We've provided a CSS classname to provide visual feedback, but this poses a problem for the visually impaired, who cannot observe the feedback when the input becomes invalid. Furthermore, validations appended to Shadow DOM cannot be accessed by screen readers unless we provide some additional context using WAI-ARIA.

While `ButtonComponent` is useable via keyboard input by virtue of extending from `HTMLButtonElement`, the addition of an SVG in the icon variant prohibits visually impaired users from understanding the context of the button element. Screen readers rely on the content of the button to provide a label, which the primary and secondary button variants handle already. The icon variant lacks a text label that screen readers can analyze and read aloud to the user.

In this section, you'll use WAI-ARIA attributes to solve two distinct problems with the form:

- Screen readers cannot analyze the button label in the icon variant
- Screen readers cannot read the validation messages so that users who rely on screen readers can understand what is needed to validate each form field

First, you'll add `ButtonComponent` to the `Form` story in `TextInput.stories.js` so that you can use `FormTemplate` as your development environment. After a brief introduction to using a screen reader during development, you'll use WAI-ARIA attributes to provide all the necessary context the screen reader requires to aid in navigating the form.

Button Form

In this chapter, you've developed `ButtonComponent` in isolation, so let's see what happens when you integrate the component into the existing Form story in `TextInput.stories.js`.

Import `ButtonComponent` into `TextInput.stories.js`.

```
1 import { ButtonComponent } from "../button/Button";
```

Swap out the `input type="submit"` in `FormTemplate` for the `button`. Set the `is` attribute to `in-button` on the button element in `FormTemplate` so that the browser can interpret the buttons as a customized built-in element. Add the appropriate classnames to the submit button. According to Figma, the submit button is a primary variant of the `ButtonComponent`.

```
1         <button class="primary" is="in-button" type="subm\
2 it">Submit</button>
```

That's it! Since `ButtonComponent` is a customized built-in element, integration was a snap. The elements in the layout appear very

close vertically and don't have the same margin between them as the mockup in Figma. To fix this issue with the layout, add a global style to `packages/style/style.css`. Select each instance of `TextInputComponent` with a CSS classname: `.form-control` and set `margin-bottom`.

```
.form-control {
  margin-bottom: var(--margin-md);
}
```

For the remainder of this chapter, use the `Form` and `Icon` stories as a test bed for providing accessibility features to `ButtonComponent` and `TextInputComponent`.

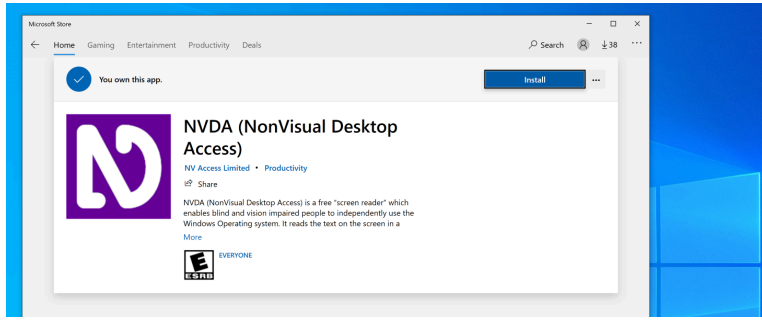
Using a screen reader

Screen readers are applications that blind, visually impaired, and dyslexic people use to interact with operating systems. Through the use of a synthesized voice and other visual cues, screen readers provide contextual information about what the user currently has focused in a user interface. NVDA is the most popular screen reader application for Windows, while VoiceOver is built into MacOS and other Apple operating systems.

If you've never used a screen reader application before, follow the steps below to install and open NVDA or VoiceOver for Windows and MacOS, respectively.

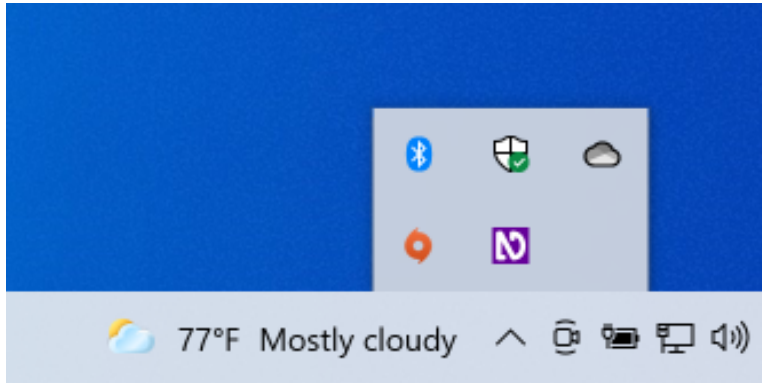
Windows

NVDA, short for NonVisual Desktop Access, is a Windows application available for free on the Microsoft Store. To install NVDA, sign in to the Microsoft Store in the Microsoft Store application.



After you have signed in to the Microsoft Store, search for NVDA, select the application in the search results, and click the **Install** button. Follow on-screen prompts to install NVDA.

Once NVDA is installed, access NVDA application preferences in the Windows Taskbar.

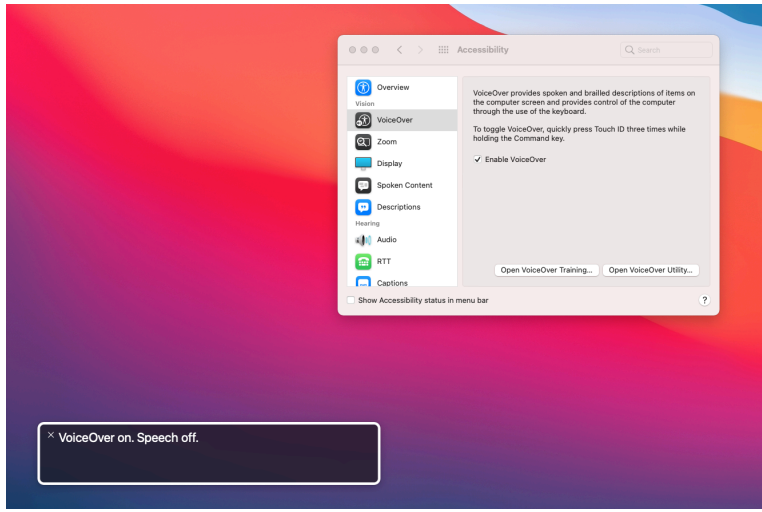


NVDA uses a synthesized voice to announce contextual information about the currently focused user interface for visually impaired and blind users in Microsoft Windows.

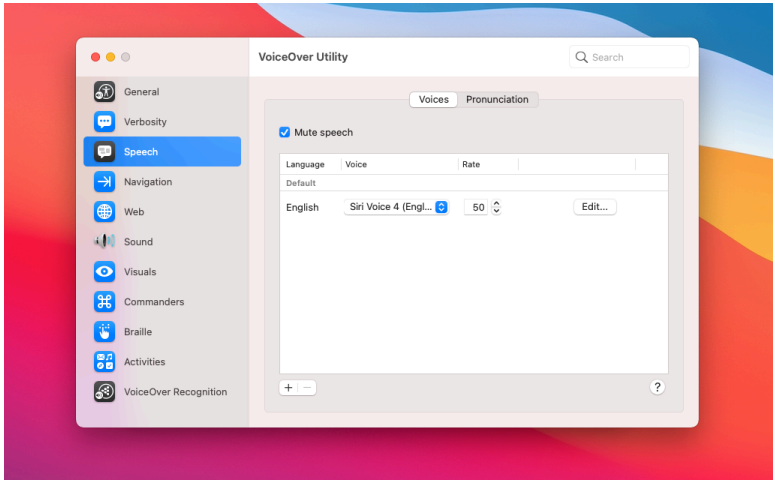
To stop NVDA at any time, click the NVDA icon in the Windows Taskbar and choose **Exit** from the dialog.

MacOS

VoiceOver is a screen reader provided by Apple Inc. in all of its operating systems. In MacOS, access VoiceOver in the System Preferences Accessibility Pane. When you click **Enable VoiceOver**, a contextual overlay appears in the lower left of the screen. VoiceOver announces what is currently in focus through a synthesized voice.



During development, you may have to change context rapidly between IDE and browser. If the synthesized voice becomes a burden, you can turn it off by clicking **Open VoiceOver Utility**... in the System Preferences Accessibility Pane. When the VoiceOver Utility Pane appears, select **Speech** in the Sidebar, and click **Mute Speech**.



Another option you may find useful during development is the ability to hide the VoiceOver cursor. In your IDE, the VoiceOver cursor displays an overlayed cursor for the visually impaired. However, this can interfere with typing for some users.

Once you have a screen reader installed, proceed to the following sections. If you've never used a screen reader before, hopefully you'll find a process for developing with a screen reader that works best for you.

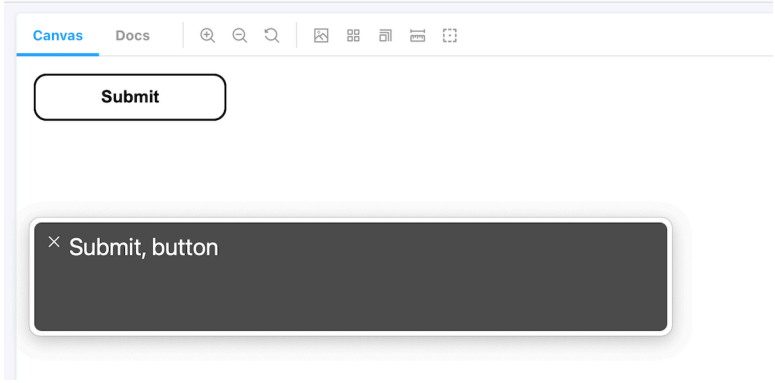
Providing context to the icon variant

Earlier in this chapter, you developed the icon variant of `ButtonComponent`. However, since the label is effectively replaced by an SVG, screen readers can no longer analyze the content of the button element.

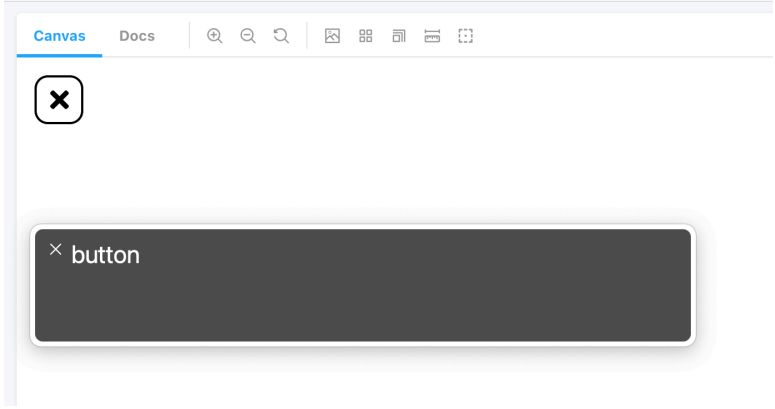
Depending on your operating system, activate NVDA or VoiceOver and leave them running when testing your changes during development for the remainder of this chapter.

If you haven't already, activate the screen reader application and

navigate to the `Primary` button story in Storybook. Click on the whitespace in the Storybook canvas to give it focus, then press the Tab key to focus the primary button. Observe how the screen reader announces the label, followed by the element name.



Navigate to the `Icon` story and focus the close button. Notice how the screen reader only provides the element because there is no contextual information that labels the button element.



To provide context for the screen reader, you'll need to use WAI-ARIA attributes, namely `aria-labelledby` and `hidden`.

`Icon.args` need to be updated to support the SVG and the label.

This change will conflict with other stories in the same file. So you don't interfere with other stories in `Button.stories.js`, make a new template for the icon variant named `IconTemplate` based off the existing `Template`.

```
const IconTemplate = ({ label, variant, svg }) => html` <\
button
```

Don't forget to update the `Icon` story, which should now reference `IconTemplate`.

```
export const Icon = IconTemplate.bind({});
```

Update `Icon.args` to support both a label and SVG, making sure the destructured properties in `IconTemplate` match the names of each property on `Icon.args`. Set the value of the `label` property to `Close`, and move the SVG to a new property named `svg`.

```
Icon.args = {
        variant: 'icon icon-close',
        label: 'Close',
        svg: svg,
};
```

Update `IconTemplate` to passing the `svg` where `label` was once in the template.

```
        ${svg}
</button>`;
```

Next, create a `<span>` tag before the `svg` and set the span element's content with the `label`. To hide the span visually, add the `hidden` attribute, which tells the browser this element should be hidden from view. Add the `id` attribute to the span element, setting the

`id` to `close-button-label`. This `id` is required for the WAI-ARIA attribute `aria-labelledby`, which needs a unique `id`.

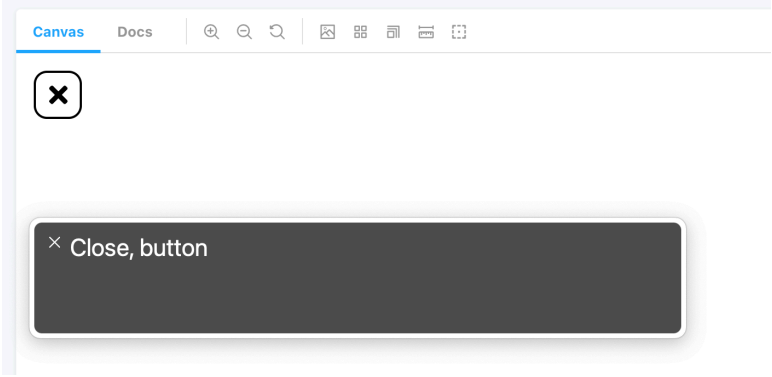
On the button element, set `aria-labelledby` to the `id` of the span, linking the two elements together for screen readers.

```
const IconTemplate = ({ label, variant, svg }) => html` <\
button
        class="${variant}"
        aria-labelledby="close-button-label"
        is="in-button"
>
        <span id="close-button-label" hidden>${label}</span>
        ${svg}
</button>`;
```

Finally, to ensure that screen readers do not analyze the SVG, set the `aria-hidden` attribute to `true` on the svg element in Button.stories.js.

```
svg.setAttribute('aria-hidden', 'true');
```

Navigate again to the Icon story in Storybook and focus the icon button with the screen reader activated. Notice how the screen reader now behaves the same way as if the button was directly labeled, like in the case of primary and secondary buttons.



`aria-labelledby` provides a pointer to an element with an `id` that contains the label for the button element. You visually hid the span element using `hidden` and used the `aria-hidden` attribute to hide the SVG from screen readers. A combination of all of these WAI-ARIA attributes allows the icon variant of `ButtonComponent` to work with a screen reader.

Providing validation messages to screen readers

The validation messages that appear when `TextInputComponent` is invalid are a custom implementation. Browsers don't have a native container for validation messages that screen readers automatically analyze. While the validation messages appear visually, we need to provide the same contextual information for the screen reader. This can be achieved through the use of WAI-ARIA attributes `aria-describedby`, `aria-role`, `aria-live`, and `aria-invalid`.

In `TextInputComponent` template, set an `id` attribute to `message` on the messages container. Link the input element to the messages container by setting the `aria-describedby` attribute to the same id: `message`. Since the messages container is just a `div`, give the screen reader more information about this element. `aria-role` set to `alert` notifies the screen reader this element contains an alert

notification, while the `aria-live` attribute set to `assertive` gives the messages contained in the div element precendence over other announcements made by the screen reader.

```
    <div class="control">
      <input type="text" aria-describedby="message" />
    </div>
    <div class="message"
         aria-role="alert"
         aria-live="assertive"
         id="message"
    ></div>
```

Navigate to the Form story and test the validations with a screen reader. Notice how the application reads back the messages. This is a huge win for accessibility because each validation message provides context for how the user can correct the form field.

We can do a little bit more work to enhance the functionality for screen readers during validation while also making the user experience for everyone else much better. Right now, when the form field is invalid, the validation messages are not hidden until the input element is blurred. If the validations were cleared as the user was typing, it would allow the visually impaired to focus on what they are typing rather than listen to the feedback of the validation messages. When the user blurs, validation will kick in and announce validation messages again, but not until then.

There's also one other problem we need to solve. Currently, the input element receives a classname `error` to show the field is invalid, but there isn't similar feedback for the screen reader. Correct this behavior by finding where the `error` classname is set in validator.ts and use the `aria-invalid` attribute set to `true` to notify screen readers the input is invalid.

```
  elem.$validator.validations.forEach((validator) => {
    if (validator.condition(elem)) {
      elem.setValidity(validator.flag, validator.message);
      if (showError) {
        if (elem.$input) {
          elem.$input.setAttribute("aria-invalid", "true"\
);
          elem.$input.classList.add("error");
        }
```

To clear out the validation messages as the user types, add a new `onkeyup` event listener in the `TextInputComponent` `connectedCallback` lifecycle. In the event listener callback, call a new method on `TextInputComponent` called `onChange`.

```
  connectedCallback() {
    this.$input.onblur = () => {
      this.onValidate(true);
    };
    this.$input.onkeyup = () => {
      this.onChange();
    };
```

In the `onChange` method, clear out the content of the messages element, remove the `error` classname and `aria-invalid` attribute from the input, and optionally, update the form with the value of the input.

```
  onChange() {
    (this.shadowRoot.querySelector(".message") as HTMLEle\
ment).innerText = "";
    this.$input.classList.remove("error");
    this.$input.removeAttribute("aria-invalid");
    this.internals.setFormValue(this.value, this.value);
  }
```

Test the new functionality in the `Form` story. When an instance of `TextInputComponent` is invalid and the user starts typing, the element should appear focused and no longer display validation messages until the user blurs and the form control is invalid. When development is complete you may turn off the screen reader.

Testing ButtonComponent

We could set up an integration test for `ButtonComponent`. However, that kind of end-to-end testing will be reserved for part three of this book when you develop a login form in the context of an application. Snapshot tests provide the most value at this point in time. After developing several interaction states for `ButtonComponent`, you can validate the styling of the button most effectively with snapshots.

Ensure Storybook is running in a Terminal window.

```
yarn storybook
```

With Storybook running, generate snapshots for the component in another Terminal window.

```
yarn test:snapshot:ci
```

When running the snapshots you'll notice a failure in the Form snapshot. Open the image file that displays the differences between the previous snapshot and current in the directory `packages/component/src/input/snapshots/diff_output`. Upon inspection, you realize the diff is because you added a bottom margin with CSS to the form controls in the layout. No harm, no foul. This change was intentional. To fix the snapshots, delete the `diff_output` directory and delete the existing snapshot.

```
rm -rf packages/component/src/input/__snapshots__/__diff_\
output__
rm packages/component/src/input/__snapshots__/web-compone\
nts-storyshots-test-js-storyshots-components-inputs-text-\
input-form-1-snap.png
```

Run the snapshot tests again. Since you deleted the offending snapshot, a new file will be generated in place.

```
yarn test:snapshot:ci
```

After snapshots have been generated for each story, add them to your commit.

Before concluding this chapter, check if any tests are broken due to the most recent changes. You did refactor the code in `packages/component/src/input/TextInput.stories.js` quite a bit to include the `HTMLButtonElement` element in the template.

```
yarn test:e2e
```

A test failure happens in the `TextInputComponent` test suite due to a timeout. The test cannot find an element with the class name `.submit`. That happened because we failed to give the `button` element this class name in packages/component/src/input/TextInput.stories.js.

```
  1) TextInputComponent
       should display error message on blur:
     AssertionError: Timed out retrying after 30000ms: Ex\
pected to find element: `.submit`, but never found it.
      at Context.eval (http://localhost:6006/__cypress/te\
sts?p=cypress/integration/TextInput.spec.js:104:36)
```

Update `FormTemplate` in `packages/component/src/input/TextInput.stories.js` with the class name the test expects.

```
<button class="primary submit" is="in-button" type="submi\
t">Submit</button>
```

Run the tests again to observe `All specs passed!`.

```
yarn test:e2e
```

Commit your changes to complete this section. Now you've created snapshots for `ButtonComponent` let's review the content of Chapter Three.

Summary

In this chapter, you learned how to implement `ButtonComponent` with customized built-in elements. By extending `HTMLButtonElement` instead of `HTMLElement`, you retained the accessibility characteristics of the native button. The primary benefit of customized built-in elements is the ability to extend existing HTML elements, thus retaining their default behaviors. In the example of the button component, this meant you didn't have to attach event listeners that handle events from keyboard input because the button element natively activates using the select or enter keys. Retaining the default behavior of native elements

comes at a cost. For those elements that cannot implement Shadow DOM, styling has to be handled via a `<style>` tag either in the `<head>`, or when applicable, the `ShadowRoot` of another custom element. In later chapters, you'll correct this behavior when you encounter it.

You also learned how form-associated custom elements participate in accessibility. Since `TextInputComponent` implemented `formAssociated`, instances of the element can be associated with `label` elements like any native form control. When you used a screen reader to navigate the form, you found the user was missing context from the validation messages found in the `ShadowRoot`. To provide appropriate validation messages to the screen reader, you implemented WAI-ARIA attributes.

So far, you've covered the three kinds of custom elements: autonomous, form-associated, and customized built-in elements. In the `FormCard` story, you've experienced how all three custom element specifications work together to provide a custom, accessible, and reusable interface. You also learned about the other technologies that comprise the set of specifications known as Web Components: Shadow DOM and HTML templates.

This marks the completion of Part One of this book. You may be thinking, "if I just learned about the full specification, why would I need to learn more?"

In Part Two, you'll handle some of the quirks and features of custom elements in a more elegant way by developing a micro-library that enables custom element development. In subsequent chapters, you'll implement more complex interface patterns with custom elements and learn how to distribute a UI component library that contains Web Components.

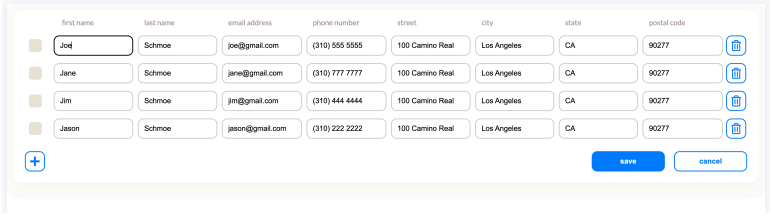
In Part Three, you'll reuse much of the code mocked in each Storybook story to develop a web application with custom elements, learning how to handle client-side routing and server-side rendering along the way.

Part Two

Part Two

UI Library

Now that you've learned about the set of specifications known as Web Components, it's time to streamline custom element development by coding a micro-library that will help you wrangle the component template, styling, and registering custom elements with the registry. After coding reusable functions to use throughout your components, you'll learn how to develop more complex component layouts with Web Components: `TableComponent`, and `DialogComponent`. Through the exercises in Chapters Five and Six, you'll expand your knowledge of HTML templates and learn different methods custom elements can use to communicate. In Chapter Seven, you'll package the custom elements you coded in previous chapters into a reusable UI component library for distribution.



What you will learn

While developing editable table, checkbox and dialog components with Web Components, you'll explore the following concepts:

- How to code reusable functions that handle common tasks in custom element development
- How to implement TypeScript decorators to streamline UI component development
- The differences between autonomous, form-associated and customized built-in elements
- How to dispatch and listen for `CustomEvent` in custom elements
- How to effectively pass data through custom element attributes
- How to code a dynamic table with custom elements
- How to architect a solution for stacking dialogs in a web application
- How to handle communication between top level and deeply nested custom elements using `BroadcastChannel`
- How to implement a separation of concerns in complex custom element layouts
- Best practices for distributing a UI library with Web Components
- How to work effectively in a monorepo that contains separate packages for a micro-library and UI library

Developing in Part Two

You'll use Storybook as the primary development environment throughout Part Two. For each chapter, run `yarn storybook` in the Terminal. Every change made to the codebase will be reflected in

the Storybook UI while Storybook is running in Terminal. When you are finished developing, cancel the server by holding down Ctrl on the keyboard and pressing C.

Micro-library

Chapter Four

Introduction

In Part One, we relied heavily on the set of specifications that make up Web Components: Custom Elements, HTML Templates, Shadow DOM, and `CustomElementRegistry`. The main benefits Web Components bring to the table: reuse, interoperable, accessible, having a long lifespan, are due to their reliance on browser specifications. Had we adopted a library or framework, we may have lost some or all of these characteristics in the user interfaces built with it. UI components coded with some libraries aren't interoperable with other JavaScript libraries or frameworks, which puts a hard limit on reuse.

Even though we gain these benefits from Web Components, we have lost any benefit JavaScript libraries have to offer. Frameworks and libraries like React, Vue, Angular and Svelte often provide an abstraction around browser specifications. Under the hood, frameworks are built with TypeScript or JavaScript, but behind the wheel engineers have an interface for component-level development. React, for example, famously opted for a purely functional approach, giving engineers "hooks" to manage side effects in user interfaces. JavaScript libraries and frameworks exist mostly to provide architectural patterns that make life easier on the part of the web developer, offering features not available with browser specifications alone, like data binding and state management. As browser standards evolve over time, the HTML living standard[53]

[53] https://html.spec.whatwg.org/multipage/

may gain some of the most popular features from JavaScript libraries. Web Components may gain more specifications that make styling components easier or enable data to be passed more efficiently and securely between components, for example.

What if we could retain the benefits of Web Components while also gaining the architectural design of a JavaScript framework? Indeed, many have tried. webcomponents.dev[54] tracks over fifty implementations of custom elements coded with a JavaScript library. This service provides performance and scalability metrics on a semi-annual basis to help inform the decision making process when selecting a library. With so many options for Web Component development, it may seem daunting to pick a library or futile to develop one on your own. It must be really difficult, right?

In this chapter we'll demystify the inner workings of Web Component libraries. Using TypeScript decorators, we'll develop a new interface that simplifies development, but doesn't compromise performance. The micro-library we'll code optimizes to less than 1Kb of minified JavaScript (optimized with brotli). Components we've already developed will transform from something like this:

```
class MyComponent extends HTMLElement {
  constructor() {
    super();
    const shadowRoot = this.attachShadow({ mode: "open" }\
);
    const template = document.createElement("template");
    template.innerHTML = `
    <style>
    :host {
      display: block;
      width: 100%;
      font-family: var(--font-default);
      font-size: var(--font-body-sm);
```

[54] https://webcomponents.dev/blog/all-the-ways-to-make-a-web-component/

```
    }
    <div class="container"></div>
    `;
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
  }
}
customElements.register("my-component", MyComponent);
```

to instead use a TypeScript decorator named `Component` like this:

```
@Component({
  selector: "my-component",
  style: css`
    :host {
      display: block;
      width: 100%;
      font-family: var(--font-default);
      font-size: var(--font-body-sm);
    }
  `,
  template: html` <div class="container"></div> `,
})
class MyComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this, { mode: "open" });
  }
}
```

Decorators are denoted by the `@` symbol, followed by the name of the decorator `function`, in this case `Component`. In the above example, the `Component function` is called with a single argument: an `Object` where the developer can declare the tag name, CSS style, and HTML template.

The side effect of using a class decorator to handle template and styling is readily apparent. Not only have we abstracted the creation of Shadow DOM, we've removed the line of code responsible for registering the `class` definition with the `CustomElementRegistry`. The `constructor` is much cleaner now. Component logic that should instantiate in the `constructor` is no longer aside template specific code. That's because all of that code has been abstracted to reusable `function` named `attachShadow`.

Decorators are found in other programming languages but currently not in the ECMAScript standard, although a proposal[55] exists. You can't use a decorator and expect it to run in the browser, at least not yet. TypeScript has supported a decorator pattern for quite some time, anticipating the time when decorators land in ECMAScript. We can use decorators in TypeScript now by configuring the `tsconfig.json` with the `experimentalDecorators` property. You don't need to change the `tsconfig.json` because the repository is already setup to work with decorators.

```
1  "experimentalDecorators": true,
```

Don't let that property spook you just because "experimental" is in the name. Several applications in production right now use TypeScript decorators. The entire Angular ecosystem has relied on TypeScript decorators since Angular 2. When TypeScript compiles down to JavaScript, a minimal amount of code is injected that polyfills the decorator pattern for ES2015 `class`.

In addition to making a `class` decorator that allows you declare tag name, styling and template with a cleaner interface than before, you'll also code a method decorator that simplifies binding event listeners to custom elements. Instead of typing `this.addEventListener('click', this.onClick)`, what if you could decorate the `onClick` method and still provide the same functionality? It could look something like this:

[55] https://github.com/tc39/proposal-decorators

```
@Listen('click')
onClick(ev: MouseEvent) {
    // do something here
}
```

If all of this seems foreign, don't fret! Coding decorators is just like coding any JavaScript `function`. Providing these framework-like features to custom elements may be easier than you think. This chapter provides step-by-step instructions for coding the decorators and applying them to the components you've already developed.

What is a micro-library?

The definition of micro-library is kind of murky because over time engineers may have used the term in different contexts. For the purposes of this chapter, let's define micro-library as a collection of prewritten code used for common development tasks that has a small footprint. Micro-libraries may have similar functionality as much larger libraries, but with way less code that optimizes down to a few Kb or maybe even less than 1 Kb. That's why they are "micro". This trend isn't by any means new. Micro-libraries have existed for awhile. Famously, Preact is a ~3 Kb alternative to the ~45 Kb React. Micro-libraries exist, in part, to provide a more performant alternative to popular JavaScript libraries. In the context of custom elements, micro-libraries are an interesting solution because we can gain the functionality of a JavaScript library with little expense to performance.

Coding in a monorepo

In Part One you mainly relied on the `yarn storybook` command for your development environment. The components you developed

so far are part of the `@in/ui` package in the packages/component directory. The package name is defined in the `package.json`. If you inspect that file you would find several `scripts` declared on the `package.json`. These `scripts` are pointers to different build tasks for the package. That's right! You've been developing in a npm package this entire time. In Chapter Seven we'll return to this `package.json` in a lesson about UI library best practices, but for now it's important you're aware this file exists.

Starting with this chapter, you're going to branch out and work in other parts of the monorepo. When developing against the micro-library, you'll code in files found in the packages/common directory. Essentially, you are going to export parts of the library for use in the `@in/ui` package. For any code you write in the `@in/common` package to be used in the `@in/ui` package, you'll need to build the `@in/common` package first, then start refactoring code in the `@in/ui` package.

This process may seem confusing at first, mainly because IDE like VS Code and Storybook may react strangely to updates in `@in/common` that are being referenced by files in the `@in/ui` package. That's due to VS Code and Storybook not interpreting dependencies in the monorepo as intuitively as we may like.

Any time you update the micro-library in the `@in/common` package and run the build, you may need to restart Storybook or potentially VS Code to force update the TypeScript AST.

Even with the quirky nature of the monorepo in regards to TypeScript compilation, we'll achieve interesting results. We'll be able to import `function` from one package into another as if the package were already published on npm.

What you will build

In this chapter you'll identify reusable parts of component and abstract logic away from each component implementation in a functional manner. You'll code a class decorator that handles declaration of a component selector, styling and template. In the next section, you'll learn how to use method decorators to attach event listeners to DOM elements. By the end of this chapter `TextInputComponent`, `ButtonComponent`, and `CardComponent` will be refactored to use the new decorator pattern. In following chapters, you'll use and build upon the micro-library where needed.

What you will learn

You'll explore the following concepts:

- How to code reusable functions that handle common tasks in custom elements
- Code `function` that appends templates to Shadow DOM
- Code `function` that handles component CSS styles
- Code `function` that handles templates to custom elements that cannot evoke Shadow DOM
- How to use a class decorator to declare the selector, template and style
- How to use a method decorator to handle event listeners
- When to apply reusable functions in everyday development

Acceptance criteria

- `TextInputComponent`, `ButtonComponent` and `CardComponent` have been refactored to use the new decorators

- `Component` and `Listen` decorators are exported from common package
- `html`, `css`, `attachShadow`, `attachStyle` and `attachTemplate` functions are exported from `@in/common` package
- All tests are passing

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-4 from where you left off at the end of chapter 3.

```
git checkout -b chapter-4
```

Class Decorator

Implementing a class decorator is going to improve the ergonomics of Web Component development quite significantly. We'll enhance the class-based implementation which custom elements rely on by cleaning up each `constructor`, choosing to move the declaration of a HTML template and CSS to a new `function` that decorates the `class`. This, combined with abstracting template and style generation to reusable `function` will have the positive effect of reducing the boilerplate for each component. For example, when you are finished with this section, `ButtonComponent` will look something like this.

```
@Component({
  custom: { extends: "button" },
  selector: "in-button",
  style: buttonStyles,
})
export class ButtonComponent extends HTMLButtonElement {
  constructor() {
    super();
  }
  connectedCallback() {
    this.classList.add("in-button");
    attachStyle(this);
  }
}
```

We're working in the `packages/common` directory of the monorepo for the first time, which currently only has one source file: `index.ts`. To keep this package organized, we should figure out a way to divide the code into logical chunks. Making a directory for all decorators seems prudent, then organizing the decorators by function.

To start developing the component decorator, create a new directory in `packages/common/src/decorator`, then make a new file at `packages/common/src/decorator/index.ts`, and another file at `packages/common/src/decorator/component.ts`.

```
mkdir -p packages/common/src/decorator
touch packages/common/src/decorator/index.ts
touch packages/common/src/decorator/component.ts
```

Any file named `index.ts` manages exports for several files in the same directory. Other files in a directory contain library code.

Open `packages/common/src/decorator/component.ts` in your IDE to begin.

Define the Interface

First let's define the interface used by the component decorator. We'll need this `interface` to keep the first argument of the decorator `function` type safe. Declare a new `interface` named `ElementMeta` and ensure it is exported with the `export` keyword.

```
1  export interface ElementMeta {
2    selector?: string;
3    style?: string;
4    template?: string;
5  }
```

There are three properties on this `interface` we should define: `selector`, `style`, and `template`. All three properties are optional, denoted by the `?` prior to `:`. Each of these properties are optional to allow flexibility for the end-user. `selector` will be used to automatically call `customElements.define` in the context of the decorator, but what if the end-user for some reason has to register components in some other way? The user could omit the selector and register the component however they like. There are times an engineer may want to style the component, or not. Alternatively, there may be no need for a template, perhaps only styling. Making each property optional allows for flexibility, but as we'll observe programmatically, we'll have to check each property on `ElementMeta` exists and react accordingly.

Component Decorator

TypeScript class decorators are factory functions, that is, they return a `function` that constructs an `Object`. The `Object` we're concerned with is the `class` definition of the component using this decorator. Based on the values of properties on the `ElementMeta`, we

want to modify the `prototype` of the `class` so we can use those properties to register the component with the `CustomElementRegistry` and instantiate a HTML template with Shadow DOM.

The basic structure of the `function` is as follows: Declare a new `function` named `Component` and make sure it's exportable with the `export` keyword. Name the first argument of the `function` `meta` and type define the argument with the interface you just created: `ElementMeta`. return a `function`. The first argument of this `function` is defined by TypeScript when you use `Component` in the context of a class decorator. `target` is the class definition of the `class` that is being decorated. Inside this factory `function`, return the class definition. `target` should be type defined as `any` here because so many different `class` will use the decorator. It would be difficult to implement type safety here, nor should we.

```
export function Component(meta: ElementMeta) {
  return (target: any) => {
    return target;
  };
}
```

What happens before the `target` is returned is the most interesting part of the class decorator. Before we start modifying the class definition, let's put a fail-safe mechanism in for users who haven't specified an `ElementMeta` in the first argument of `Component`.

```
export function Component(meta: ElementMeta) {
  if (!meta) {
    console.error("Component must include ElementMeta to \
compile");
    return;
  }
```

The above conditional checks if `meta` is `undefined` and when truthy,

displays a console error for developers in Dev Tools, then aborts the rest of the algorithm that depends on `ElementMeta`.

We have two objectives for the class decorator: register the component using the `selector` property, then add metadata on the `class` definition to later append a HTML template to Shadow DOM using the `style` and `template` properties on the `ElementMeta`.

Let's start with registering the component via the decorator function. In the factory function, reference the `selector` and pass it and the class definition, here named `target`, to `customElements.define`.

```
export function Component(meta: ElementMeta) {
  if (!meta) {
    console.error("Component must include ElementMeta to \
compile");
    return;
  }
  return (target: any) => {
    customElements.define(meta.selector, target);
    return target;
  };
}
```

This one line allows the end-user to omit a call from `customElements.define` in their implementation, as you'll later observe when refactoring `TextInputComponent`, `ButtonComponent`, and `CardComponent`, but the user has the option to ignore `selector` entirely in the `ElementMeta`, so be sure to wrap the call to `define` in a conditional.

```
if (meta.selector) {
  customElements.define(meta.selector, target);
}
```

That was easy enough, although we're not yet using the class decorator for its main function: modifying the `prototype` of the

`class` it decorates. This just happens to be a convenient place to also make the call to `customElements.define`. We can't use this space to call `attachShadow` and create a HTML template, then append it to the `shadowRoot` like we've done in previous chapters, because the class decorator doesn't have access to the instance of the class, only the definition.

Class decorators are a place where you can modify the `prototype` of a `class`, so that's exactly what we're going to do. We're going to store the `ElementMeta` passed through the `Component function` arguments on the `prototype` of the `class`, so we can later access the `template` and `style` in the `constructor` of the component `class`.

Set a new property called `elementMeta` on the `prototype` to the `meta` that was passed into the `Component function`.

```
target.prototype.elementMeta = meta;
```

If we want to avoid the `style` and `template` properties from ever being `undefined`, here is a good place for some conditionals that set both of these properties to empty `string` in the case they are `undefined`.

```
if (!meta.style) {
  meta.style = "";
}
if (!meta.template) {
  meta.template = "";
}
```

When you are finished with the `Component` decorator, the `function` should look like this:

```
export function Component(meta: ElementMeta) {
  if (!meta) {
    console.error("Component must include ElementMeta to \
compile");
    return;
  }
  return (target: any) => {
    if (!meta.style) {
      meta.style = "";
    }
    if (!meta.template) {
      meta.template = "";
    }
    target.prototype.elementMeta = meta;
    if (meta.selector) {
      customElements.define(meta.selector, target);
    }
    return target;
  };
}
```

Build the `@in/common` package

Development of the `Component` decorator is finished, but you still need to provide the decorator and possibly the `interface` for use outside of the `@in/common` package. In previous chapters you developed components in a sibling directory at `packages/component`. To access the `Component` decorator from the files in that package, you need to properly `export` from the `@in/common` package.

To finalize development of the decorator, add the proper exports to `packages/common/src/decorator/index.ts` and `packages/common/index.ts`.

In `packages/common/src/decorator/index.ts` add the following exports.

```
export { Component } from "./component";
export type { ElementMeta } from "./component";
```

Export the same from `packages/common/index.ts`.

```
export { Component } from "./src/decorator";
export type { ElementMeta } from "./src/decorator";
```

`index.ts` is the entry point of the `@in/common` package. When the `@in/common` package is built, other packages in the monorepo will be able to reference anything exported from the `index.ts`.

Check the build successfully runs, which means your TypeScript is valid. This is your first time running the `build` command because previously you didn't really need to. Storybook was handling the development environment for all the components so far. In future chapters you'll need to build the `@in/ui` package as well, but for now you'll need to make sure the `@in/common` package is built using the following command.

```
lerna exec --scope @in/common -- yarn build
```

If you get an error like `zsh: command not found: lerna` make sure lerna is installed globally and try the build command again.

```
npm install --global lerna
```

If you see `lerna success` output in the Terminal, you are ready to proceed. Otherwise, check your TypeScript is correct.

Abstracting attachShadow

Now that we have `ElementMeta` stored on the `prototype` of any `class` using the `Component` decorator, we can write a reusable

`function` that could be used in the `constructor` of the same `class` to instantiate Shadow DOM much like we already did in the components developed in Part One. By abstracting this logic to a reusable `function` we can reduce several lines of code in each component implementation down to one line.

Basically we want to take something like this...

```
const shadowRoot = this.attachShadow(options);
const template = document.createElement("template");
template.innerHTML = `
    <style>
      :host {
        display: block;
        background: var(--color-white);
        border-radius: var(--radius-md);
        box-shadow: var(--shadow);
        overflow: hidden;
        max-width: 320px;
      }
    </style>
    <div class="container"></div>
    `;
shadowRoot.appendChild(template.content.cloneNode(true));
```

and reduce it to one line.

```
attachShadow(this, options);
```

The first argument of `attachShadow` is the instance of the `class` which in the `constructor` you can reference as `this`. The second argument is the `Object` that configures the call to `element.attachShadow`. According to MDN[56], this `Object` can accept three fields: `mode` and `delegatesFocus`. The type definition

[56] https://developer.mozilla.org/en-US/docs/Web/API/Element/attachShadow

exported from TypeScript's lib.dom.d.ts[57] called `ShadowRootInit` reveals a third field: `slotAssignment`. You've set the fields on this `Object` in prior chapters for `TextInputComponent`, `ButtonComponent`, and `CardComponent`.

To start development of this new `function`, make a new directory named template in `packages/common/src` and create a new file in that directory named `index.ts`. Create another file in the directory named `shadow.ts`.

```
mkdir packages/common/src/template
touch packages/common/src/template/index.ts
touch packages/common/src/template/shadow.ts
```

In `packages/common/src/template/shadow.ts`, create a new `function` named `attachShadow` and `export` it from the file. Declare two arguments for `attachShadow`: `context` and `options`, making `options` optional with `?` and type defining `context` as `any` for now and `options` as `ShadowRootInit`, a type definition exported from lib.dom.d.ts.

```
export function attachShadow(context: any, options?: Shad\
owRootInit) {}
```

Follow up in `packages/common/src/template/index.ts` and ensure `attachShadow` is exported for the main `index.ts`.

```
export { attachShadow } from "./shadow";
```

Finally in `packages/common/index.ts` export the `attachShadow` `function`.

[57] https://github.com/microsoft/TypeScript/blob/97a7901f26290a13ea6c3b057c8720c1614cb221/lib/lib.dom.d.ts#L1610

```
export { attachShadow } from "./src/template";
```

Jumping back to `packages/common/src/template/shadow.ts`, fill in the algorithm for `attachShadow`, referencing the `constructor` in `ButtonComponent` at `packages/component/src/button/Button.ts` as a guide. Make a `const` named `shadowRoot`, type defined as `ShadowRoot`, equal to `context.attachShadow`. Since we've opted to make `options` optional, we need to at the very least give `context.attachShadow` the minimum `ShadowRootInit`, which requires `mode`.

```
const shadowRoot: ShadowRoot = context.attachShadow(
  options || { mode: "open" }
);
```

On the next line, make a `const` named `template` equal to `document.createElement('template')`. This line creates a new HTML template.

```
const template = document.createElement("template");
```

Set the content of the HTML template using the `ElementMeta` stored on the `prototype` of whatever `class` will use this `attachShadow function`. Pass in `context.elementMeta.style` to a `<style>` tag and `context.elementMeta.template` afterward.

```
template.innerHTML = `<style>${context.elementMeta.style}\
</style>${context.elementMeta.template}`;
```

Finally, append a clone of the HTML template to the `ShadowRoot`.

```
shadowRoot.appendChild(template.content.cloneNode(true));
```

When you are finished, the `attachShadow function` should look like this:

```
export function attachShadow(context: any, options?: Shad\
owRootInit) {
  const shadowRoot: ShadowRoot = context.attachShadow(
    options || { mode: "open" }
  );
  const template = document.createElement("template");
  template.innerHTML = `<style>${context.elementMeta.styl\
e}</style>${context.elementMeta.template}`;
  shadowRoot.appendChild(template.content.cloneNode(true)\
);
}
```

With `Component` and `attachShadow` now supporting autonomous and form-associated custom elements, you can now refactor `CardComponent` and `TextInputComponent` to use the new decorator pattern. Build the `@in/common` package again so files inside the `@in/ui` package can pickup the latest changes.

```
lerna exec --scope @in/common -- yarn build
```

Refactor CardComponent and TextInputComponent

Open `packages/component/src/card/Card.ts` and observe the current state of the `CardComponent class`. Before you replace the content in the `constructor`, import `Component` and `attachShadow` from `@in/common`.

```
import { Component, attachShadow } from "@in/common";
```

Add the `Component` decorator to `CardComponent`.

```
@Component({

})
export class CardComponent extends HTMLElement {
```

Remove the line at the bottom of the file that registers the component, noting the tag name `in-card`.

Remove `customElements.define('in-card', CardComponent);`.

Set the `selector` property to the `ElementMeta` passed into `Component` to `in-card`, the same `string` originally used to register the component.

```
@Component({
  selector: 'in-card',
})
```

Carefully move the content of the `<style>` tag in the `constructor` to the new `style` property on `ElementMeta`.

```
@Component({
  selector: 'in-card',
  style: `
  :host {
    display: block;
    background: var(--color-white);
    border-radius: var(--radius-md);
    box-shadow: var(--shadow);
    overflow: hidden;
    max-width: 320px;
  }
  ::slotted(*) {
    padding-left: var(--padding-lg);
    padding-right: var(--padding-lg);
  }
```

```
  ::slotted(a:link), ::slotted(a:visited) {
    display: block;
  }
  ::slotted(:last-child) {
    padding-bottom: var(--margin-lg);
  }
  ::slotted(img) {
    width: 100%;
    padding-left: 0px;
    padding-right: 0px;
  }
  `,
})
```

Do the same for the template of `CardComponent`. Carefully migrate the HTML to the new `template` property until the `ElementMeta` is filled in.

```
@Component({
  selector: 'in-card',
  style: `
  :host {
    display: block;
    background: var(--color-white);
    border-radius: var(--radius-md);
    box-shadow: var(--shadow);
    overflow: hidden;
    max-width: 320px;
  }
  ::slotted(*) {
    padding-left: var(--padding-lg);
    padding-right: var(--padding-lg);
  }
  ::slotted(a:link), ::slotted(a:visited) {
    display: block;
```

```
  }
  ::slotted(:last-child) {
    padding-bottom: var(--margin-lg);
  }
  ::slotted(img) {
    width: 100%;
    padding-left: 0px;
    padding-right: 0px;
  }
  `,
  template: `
    <header>
      <slot name="header"></slot>
    </header>
    <section>
      <slot name="content"></slot>
    </section>
    <footer>
      <slot name="footer"></slot>
    </footer>
  `,
})
```

Remove everything in the `constructor` and replace it with a call to `attachShadow`, passing in `this` to the first argument.

```
export class CardComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

If you don't already have Storybook running, start up the development server and observe the changes in the browser. Nothing

should have changed about the user interface. Everything should appear the same.

```
yarn storybook
```

Open `packages/component/src/input/TextInput.ts` and refactor the same parts of `TextInputComponent`. To start, import `Component` and `attachShadow` from `@in/common`.

```
import { Component, attachShadow } from "@in/common";
```

Add the `Component` decorator to `TextInputComponent`.

```
@Component({

})
export class TextInputComponent extends HTMLElement {
```

Remove the line at the end of the file where the `class` is registered with the `CustomElementRegistry`, taking note of the tag name `in-textinput`. Add the same `string` as the value for the `selector` property on the `ElementMeta`.

```
@Component({
  selector: 'in-textinput',
})
```

Move the styling and template to `style` and `template` properties in `ElementMeta`, respectively. When you are finished moving the template, remove the legacy code in the `constructor`, add a call to `attachShadow`, making sure to retain the call to `attachInternals` for the form-associated custom element. When you are done, the decorator and `constructor` should appear like this:

```
@Component({
  selector: 'in-textinput',
  style: `
  :host {
    display: block;
    font-family: var(--font-default);
    font-size: var(--font-body-sm);
  }
  input {
    height: var(--input-min-dimension);
    width: 100%;
    border-radius: var(--radius-sm);
    border: var(--border-default);
    font-size: var(--font-body-md);
    padding-left: var(--padding-sm);
    outline: none;
    box-sizing: border-box;
  }
  input:focus,
  input:focus:hover,
  input:active {
    border: var(--border-focus);
  }
  input.error,
  input.error:hover,
  input.error:focus,
  input.error:active {
    border: var(--border-error);
    outline: none;
    box-shadow: none;
    color: var(--color-error);
  }
  .message {
    margin-top: var(--margin-xxs);
    color: var(--color-error);
```

```
36      font-weight: var(--font-weight-default);
37    }
38    input[disabled] {
39      opacity: var(---color-disable);
40      background: var(--color-disable);
41      border: var(--border-disable);
42    }
43    input[disabled]:hover,
44    input[disabled]:focus,
45    input[disabled]:active {
46      border: var(--border-disable);
47      outline: none;
48      box-shadow: none;
49    }
50    `,
51    template: ` <div class="control">
52        <input type="text" aria-describedby="message" />
53      </div>
54      <div
55        class="message"
56        aria-role="alert"
57        aria-live="assertive"
58        id="message"
59      ></div>`,
60  })
61  export class TextInputComponent extends HTMLElement {
62    static formAssociated = true;
63    private attachInternals: () => ElementInternals;
64    private internals: ElementInternals;
65    public $validator: Validator;
66    private $attr = {};
67    constructor() {
68      super();
69      attachShadow(this);
70      this.internals = this.attachInternals();
```

```
71    }
72  ...
```

That's it! You're done refactoring `TextInputComponent` and `CardComponent` when the components appear to work in Storybook as they did prior to the refactor. So far we've only supported autonomous and form-associated custom elements in the `Component` decorator. Let's explore how to add support for customized built-in elements and follow up with refactoring `ButtonComponent` to use the new decorator.

Customized built-in elements

Supporting customized built-in elements in the `Component` decorator is actually fairly trivial, partly because we separated template logic (`attachShadow`) from the logic that registers the component with the `CustomElementRegistry` in `Component`. If you recall from Chapter 3, the only real difference between registering a customized built-in element is the third argument in the call to `customElements.define`. This `Object` is called `ElementDefinitionOptions` in lib.dom.d.ts[58].

```
1  customElements.define("in-button", ButtonComponent, { ext\
2  ends: "button" });
```

The template logic is more of a challenge. In `ButtonComponent` we included logic to add a stylesheet for the component styles to the `document.head` when the element was instantiated on the page.

[58] https://github.com/microsoft/TypeScript/blob/97a7901f26290a13ea6c3b057c8720c1614cb221/lib/lib.dom.d.ts#L431

```
  addStylesheet() {
    const head = document.head;
    if (document.getElementById(`in-button-style`)) {
      return;
    }
    const style = document.createElement('style');
    style.setAttribute('id', `in-button-style`);
    style.textContent = buttonStyles;
    head.appendChild(style);
  }
```

It would be a shame if we had to add this logic for every customized built-in element we wanted to develop. Similar to `attachShadow`, we'll create a reusable `function` called `attachStyle` that appends the stylesheet where it needs to be located in DOM to effectively style the instance of the element. For elements that want to handle a template without Shadow DOM, we'll make a `function` named `attachTemplate` that appends a template to the element instance.

ElementDefinitionOptions

Go ahead and add a new property to `ElementMeta` in `packages/common/src/decorator/component.ts` named `custom` and type define it as `ElementDefinitionOptions`. Make sure this new property is optional by declaring it with `?`.

```
export interface ElementMeta {
  custom?: ElementDefinitionOptions;
  selector?: string;
  style?: string;
  template?: string;
}
```

In the `Component` decorator, update the logic surrounding the call to `customElements.define`. Since the user can inject

`ElementDefinitionOptions` through the `ElementMeta` and it's optional, we'll need to check if `meta.custom` exists and react accordingly. This is mostly architected this way due to the fact `ElementDefinitionOptions` has a required property: `extends`.

```
    if (meta.selector && !meta.custom) {
      customElements.define(meta.selector, target);
    } else if (meta.selector && meta.custom) {
      customElements.define(meta.selector, target, meta.c\
ustom);
    }
```

attachStyle and attachTemplate

With the absence of Shadow DOM in most elements, we'll add new functions for handling the template and styling. So far, we have no need for a `function` that appends a template to an element, but we'll create one anyways, anticipating the need for one in the future.

The style sheet is another problem.

Appending a style sheet to the custom element's template without Shadow DOM poses a host of problems, so we opted to add the style tag to the `document.head` before. That implementation has another problem we haven't encountered yet. Suppose there is a customized built-in element appended to the Shadow DOM of an autonomous custom element. Shadow DOM would prohibit the styles located in the `<style>` tag in `document.head` to penetrate the shadow boundary and style the customized built-in element. We've observed only CSS variables can penetrate the shadow boundary thus far. How could we work around this problem? Luckily we always know where the instance of the customized built-in element exists in DOM. We can write an algorithm that walks up the DOM tree, checking for shadow boundaries, and if the algorithm encounters one, append the `<style>` tag there, otherwise inject the style sheet into the `document.head` like we did before.

`getRootNode`[59] is a method that exists on HTML elements to solve problems like this one. Calling `getRootNode()` returns the element's known root, which could be a `ShadowRoot` or `document`. We need a `function` that returns the nearest root node so we can append the `<style>` tag to it.

In `packages/common/src/template` make a new file named `style.ts`.

```
touch packages/common/src/template/style.ts
```

In this new file, make a new `function` called `closestRoot`. This `function` takes one argument we'll call `base`, and type define that argument as `Element`. `closestRoot` will return a `Node`, but unfortunately there isn't a type definition in `lib.dom.d.ts` that can satisfy how we need to interact with the `Node`, so we'll opt for `any` here.

```
function closestRoot(base: Element): any {}
```

Write an algorithm that checks if the `base.getRootNode()` equals `document` and if truthy return `document.head`, the `Node` we want to append the `<style>` tag. Next, check if `base.getRootNode()` is truthy, that means the algorithm has found a `ShadowRoot`, and return the `Node` returned by `base.getRootNode()`. Finally, return `document.head` if neither condition is satisfied. When you are done, the `closestRoot function` should be as follows:

[59] https://developer.mozilla.org/en-US/docs/Web/API/Node/getRootNode

```
function closestRoot(base: Element): any {
  if (el.getRootNode() === document) {
    return document.head;
  } else if (el.getRootNode()) {
    return el.getRootNode();
  } else {
    return document.head;
  }
}
```

Next, make a `function` named `attachStyle`, which takes one argument named `context`, type defined as `any`. This `function` doesn't return anything so you can type define the return as `void`.

```
function attachStyle(context: any): void {}
```

`attachStyle` needs to create a `<style>` tag, set the content of the element with the `style` property from `elementMeta` and then append the `<style>` tag to the closest root node. In `attachStyle`, make a new `const` called `closest` equal to `closestRoot(context)`. Make another `const` named `template` and create a new `HTMLStyleElement` with it. Set the `innerText` of `template` to `context.elementMeta.style`. Finally, append the `template` to `closest`.

```
function attachStyle(context: any): void {
  const closest = closestRoot(context);
  const template = document.createElement("style");
  template.innerText = context.elementMeta.style;
  closest.appendChild(template);
}
```

That would work if there was one instance of the element. That element would call `attachStyle`, appending the styling it requires to display correctly. If there are many instances of the element

though, a `<style>` tag would get appended for each instance, so we need to find a way to mitigate that. One solution is to get the `HTMLStyleElement` a unique id and check if the element exists before appending another. We have a unique identifier of sorts: the `selector` property on `ElementMeta`. Use that to set an `id` attribute on the `template`, appending another `string` to the `selector`. `-in-style` should work. This reduces the chances of any two elements having the same unique `id` attribute.

```
function attachStyle(context: any): void {
  const id = `${context.elementMeta.selector}`;
  const closest = closestRoot(context);
  const template = document.createElement("style");
  template.setAttribute("id", `${id}-in-style`);
  template.innerText = context.elementMeta.style;
  closest.appendChild(template);
}
```

Next, implement two checks to determine if the `HTMLStyleElement` already exists. If `closest` is equal to `document.head`, the element will have a `tagName` of `HEAD`. There we can query the `document` for the `HTMLStyleElement`. If `closest` has the method `getElementById`, it's an `Element` near a shadow boundary, where we can implement a similar check. If either condition returns truthy, `return` so that any new `HTMLStyleElement` isn't created. When you are finished, `attachStyle` should appear like below.

```
function attachStyle(context: any): void {
  const id = `${context.elementMeta.selector}`;
  const closest = closestRoot(context);
  if (closest.tagName === "HEAD" && document.getElementBy\
Id(`${id}-in-style`)) {
    return;
  }
  if (closest.getElementById && closest.getElementById(`$\
{id}-in-style`)) {
    return;
  }
  const template = document.createElement("style");
  template.setAttribute("id", `${id}-in-style`);
  template.innerText = context.elementMeta.style;
  closest.appendChild(template);
}
```

When you are finished coding `attachStyle`, ensure it is exported from `style.ts`.

```
export function attachStyle(context: any): void {
```

Update `packages/common/src/template/index.ts` with the new export.

```
export { attachShadow } from "./shadow";
export { attachStyle } from "./style";
```

Finally export `attachStyle` from packages/common/index.ts.

```
export { Component } from "./src/decorator";
export type { ElementMeta } from "./src/decorator";
export { attachShadow, attachStyle } from "./src/template\
";
```

Before you can refactor `ButtonComponent` to use the new `attachStyle function` and `Component decorator`, you need to build the `@in/common` package with the latest.

```
lerna exec --scope @in/common -- yarn build
```

Refactor ButtonComponent

It's time to refactor `ButtonComponent` to use the new class decorator and `attachStyle function`. Make sure Storybook is running. If you had Storybook running while editing files in the `@in/common` package, you may need to stop Storybook with Control + C in your Terminal, cancelling the process, and then restarting the process with `yarn storybook`. Sometimes Storybook can get mixed up when packages in a monorepo are updated while Storybook is running.

Open `packages/component/src/button/Button.ts` and begin by importing the decorator and `function`.

```
import { attachStyle, Component } from "@in/common";
```

Remove the line at the end of the file that calls `define` on the `CustomElementRegistry`, but make note of the tag name in the first argument and the third argument passed to `define` that you'll need to declare in the `Component` decorator.

```
customElements.define("in-button", ButtonComponent, { ext\
ends: "button" });
```

Add the `Component` decorator to the `ButtonComponent class`, filling in the necessary properties: `custom`, `selector` and `style`. You may opt to keep the `buttonStyles const` or set the `string` directly on `style`. `selector` should equal `in-button`, the tag name previously passed to `customElements.define`. `custom` should be set to `{ extends: 'button' }`, the same as it was before.

```
@Component({
  custom: { extends: 'button' },
  selector: 'in-button',
  style: buttonStyles,
})
export class ButtonComponent extends HTMLButtonElement {
```

Remove the call to `this.addStyleSheet()` in the `connectedCallback` and remove the `addStyleSheet` method. In the `connectedCallback`, add a call to `attachStyle`, passing in `this` as the context for the `function`. When you are finished, the `ButtonComponent class` should appear like this.

```
@Component({
  custom: { extends: "button" },
  selector: "in-button",
  style: buttonStyles,
})
export class ButtonComponent extends HTMLButtonElement {
  constructor() {
    super();
  }
  connectedCallback() {
    this.classList.add("in-button");
    attachStyle(this);
```

```
13    }
14  }
```

Check Storybook to ensure all variants of `ButtonComponent` are displaying properly and then, in another Terminal window, run the tests. There shouldn't be any test failures because we didn't change any source code, although we did just heavily refactor each component to use the micro-library. It's probably a good idea to check nothing broke.

```
1  yarn test:snapshot:ci
2  yarn test:e2e:open
```

Coding a generic attachTemplate

We have coverage for every template so far with `attachShadow` and `attachStyle`. These functions handle CSS styles scoped to any component implementation whether or not the component relies on Shadow DOM. `ButtonComponent` uses `attachStyle`, but doesn't employ it's own custom template. Shadow DOM is only supported by a handful of DOM elements[60]. For the rest we may need a way to append a template directly to the element using the `Component` decorator. In this section, you'll code a simple `function` named `attachTemplate` that appends a HTML template to the element.

Make a new file named `template.ts` in the `packages/common/src/template` directory.

```
1  touch packages/common/src/template/template.ts
```

In this file, export a new `function` named `attachTemplate`. This `function` takes one argument, the `context`, i.e. the element the

[60] https://developer.mozilla.org/en-US/docs/Web/API/Element/attachShadow

template will be appended. `context` could be any element in DOM, so type define the argument as `any`. `attachTemplate` doesn't return anything, so the return type is `void`.

```
export function attachTemplate(context: any): void {}
```

`attachTemplate` should create a HTML template, set the template `innerHTML` to the content of the `ElementMeta` `template` property, then append a clone of the template to the `context`. When completed, the `function` should be as follows:

```
export function attachTemplate(context: any): void {
  const template = document.createElement("template");
  template.innerHTML = context.elementMeta.template;
  context.appendChild(template.content.cloneNode(true));
}
```

For some, this implementation may seem trivial, it simply takes a `string` and sets the content of the template, then appends that content to the element. This implementation could get more complicated over time. Perhaps the content needs to be injected into a child element or prepended before all other content in the element. Maybe the end-user wants to reference a HTML template already available in DOM, so we don't need to create a new HTML template. These are valid use cases we may explore later, but for now we just need a `function` in the micro-library that can handle a very simple use case because the `Component` decorator supports a template string for any element, not just those that can use Shadow DOM.

To finish this implementation, ensure `attachTemplate` is properly exported by the `@in/common` package.

Export `attachTemplate` from `packages/common/src/template/index.ts`.

```
export { attachShadow } from "./shadow";
export { attachStyle } from "./style";
export { attachTemplate } from "./template";
```

Export `attachTemplate` from `packages/common/index.ts`.

```
export { Component } from "./src/decorator";
export type { ElementMeta } from "./src/decorator";
export { attachShadow, attachStyle, attachTemplate } from\
 "./src/template";
```

When you are finished updating the exports, build the `@in/common` package.

```
lerna exec --scope @in/common -- yarn build
```

Enhancing developer experience with html and css

You've completed the necessary tasks to provide a micro-library that handles the component selector, styling, and template. Each component looks much cleaner since refactoring them to use the new `Component` decorator. There's still one problem to solve though. Our IDE has no way to detect the strings setting `style` and `template` in the decorator `function` are CSS and HTML. This means no syntax highlighting, which developers have come to expect.

```
@Component({
  selector: 'in-card',
  style: `
  :host {
    display: block;
    background: var(--color-white);
    border-radius: var(--radius-md);
    box-shadow: var(--shadow);
```

It is possible to gain syntax highlighting in the template strings with a little help from simple `function` and the proper extension for your IDE. es6-string-html[61] is an extension for VS Code that adds syntax highlighting support for multi-line JavaScript template strings. After installing this extension, nothing happens at first to `CardComponent`, `TextInputComponent`, or `ButtonComponent`. If you poke around a little, you may notice the template returned by `FormTemplate` in `packages/component/src/input/TextInput.stories.js` has HTML that is syntax highlighted. That's due to the template string being set as an argument on the `html function` imported from `lit-html`. This `function` has some superpowers, like being able to interpet `@event` attribute syntax, but we haven't yet chosen to adopt lit-html in the component library. As alluring as that may be, the micro-library we just developed will produce a smaller bundle than lit-html. We'd lose that benefit. For some people this may also be confusing, because isn't a function supposed to be called with parenthesis like `html()`? It turns out there's an odd quirk with JavaScript that allows the `html function` to be called with a single argument, the template string sitting directly beside `html` This is known as a tagged template literal and has been around since ES2015.

[61] https://marketplace.visualstudio.com/items?itemName=Tobermory.es6-string-html

In addition to the VS Code extension, we'll need two `functions` exported from `@in/common` that let the extension interpret the template strings as HTML and CSS. All that is really needed for the extension to work is the keyword `html` or `css` prior to the multi-line string.

```
return html`
  <form
    name="foo"
    slot="content"
```

In `packages/common/src/template/template.ts`, export two new `functions` named `css` and `html`. These two `functions` will handle tagged template literals for CSS and HTML templates. As tagged template literals, both `functions` should take a `TemplateStringsArray` named `template` in their first argument and return a `string`. `template` must be properly typed `TemplateStringsArray` because tagged template literals pass as an immutable array-like `Object` TypeScript defines as `TemplateStringsArray`. The second argument is a bit more ambiguous. Template literals in JavaScript can accept variable members used for string interpolation, with the notation `${}`. `TemplateStringsArray` is an `Array` of `string` separated by any instances of string interpolation. These instances are found in another `Array` that can be spread from the second argument. The items in this `Array` are ambiguous because they could be anything; a `string`, `number`, or `function`. If the engineer using these functions were to use string interpolation, you would need some special handling for this in `html` and `css`, so loop over the `TemplateStringsArray` and concatenate the same index of an item in the `rest Array` in every loop, handling the case where there is no such item. Finally, return the `string`. Make sure to export both `html` and `css`.

```
export function html(template: TemplateStringsArray, ...r\
est: any): string {
  let str = "";
  template.forEach((string, i) => {
    str += `${string} ${rest[i] || ""}`;
  });
  return str;
}

export function css(template: TemplateStringsArray, ...re\
st: any): string {
  let str = "";
  template.forEach((string, i) => {
    str += `${string} ${rest[i] || ""}`;
  });
  return str;
}
```

Follow through, exporting `html` and `css` in `packages/common/src/template/index.ts`

```
export { attachShadow } from "./shadow";
export { attachStyle } from "./style";
export { attachTemplate, html, css } from "./template";
```

Export `html` and `css` from packages/common/index.ts.

```
export { Component } from "./src/decorator";
export type { ElementMeta } from "./src/decorator";
export {
  attachShadow,
  attachStyle,
  attachTemplate,
  html,
  css,
} from "./src/template";
```

When you are finished updating the exports, build the `@in/common` package.

```
lerna exec --scope @in/common -- yarn build
```

In `packages/component/src/card/Card.ts` import both `html` and `css` from `#in/common` and apply each `function` to the `template` and `style` properties, respectively.

```
import { Component, attachShadow, html, css } from "@in/c\
ommon";
```



Repeat the process for `TextInputComponent` in `packages/component/src/input/TextI`
In `ButtonComponent` you'll only need to import `css` and use it to

provide syntax highlighting for the inline CSS. When all three components now display syntax highlighting for the `template` and `style` properties, your work is completed.

We've made some extraordinary progress with fairly little code exported from `@in/common` to support component development. There is so much functionality we could provide from the micro-library, but at the expense of bundle size and performance. Keeping the component implementation close to the metal, as they say, is beneficial for performance. Sticking to browser specification while providing some syntactical sugar for developers as they work with spec is the best of both worlds. In the next section, we'll expand the micro-library with one more feature before swtiching back to component development: a method decorator that handles callbacks passed to `addEventListener`.

Method Decorator

In the last section, you refactored each component to use the `Component` class decorator. That cleaned up the `constructor` of each component `class` fairly nicely. Another method on the `class` that has started to get polluted with logic is the `connectedCallback`. In `TextInputComponent`, the `connectedCallback` features several event listeners that bind each `function` callback to a `class` method using arrow brackets to ensure the context (`this`) is the `class` and not `this.$input`.

```
connectedCallback() {
  for (let prop in this.$attr) {
    this.$input.setAttribute(prop, this.$attr[prop]);
  }
  this.$input.onblur = () => {
    this.onValidate(true);
  };
  this.$input.onchange = () => {
    this.onChange();
  };
  this.$input.onkeyup = () => {
    this.onChange();
  };
  this.onValidate(false);
}
```

Binding event listeners to `function` callbacks is fairly common practice in `connectedCallback` because the component template can be reliably queried in this lifecycle method. Listening for events is fairly common practice in web development, so perhaps the micro-library could provide a better interface for binding `class` methods to event listeners. Enter method decorators. In the example below, a method decorator named `Listen` augments the behavior of `onValidate`. `Listen` is a `function` that takes two arguments: the `Event` name and optionally, a selector that could be passed to `querySelector` to bind the event to an element using `addEventListener`. This all happens behind the scenes now. `Listen` has direct access to the method it decorates, so we can pass the method to the second argument of `addEventListener` making the `class` method the callback that is called when the `Event` happens.

```
  @Listen('blur', 'input')
  onValidate() {
    validate(this, true);
  }
```

Method decorators are composible. If the same callback applies to two different event listeners, you can stack the decorators on the method and expect both to work just fine.

```
@Listen('change', 'input')
@Listen('keyup', 'input')
onChange() {
  this.shadowRoot.querySelector('.message').innerHTML = '\
';
  this.$input.classList.remove('error');
  this.$input.removeAttribute('aria-invalid');
  this.internals.setFormValue(this.value, this.value);
}
```

If we were to implement the method decorator pattern, this would virtually eliminate the need to bind event listeners in `connectedCallback`, freeing up the method for other logic.

```
connectedCallback() {
  for (let prop in this.$attr) {
    this.$input.setAttribute(prop, this.$attr[prop]);
  }
  validate(this, false);
}
```

Method decorators are arguably more statically analyzable as well, given that engineers have the option of calling methods on the element (`onchange`) or using `addEventListener('change')` for event listening. A method decorator standardizes the practice

of coding for events. Developers can forget to remove event listeners when an element is removed from DOM. To be safe, you have to call `removeEventListener` on the `disconnectedCallback`, passing in the `function` callback to remove the event listener. That is another potential flaw method decorators can remedy. When coding the `Listen` decorator you'll observe that we still bind event listeners in much the same way, along with the `connectedCallback`, but we can also automate removal of event listeners in the `disconnectedCallback`, all without interfering with an engineer's code.

Method decorators, unlike class decorators, have access to the `class` instance. When modifying the `class` with a decorator you only had access to the `prototype`, but with method decorators you can expect the context (`this`) to point to the instance of the `class`. Method decorators can deconstruct the `class` and do things with the `prototype`, but for reliability the safest place to modify a `prototype` is the class decorator.

For the rest of this section, you'll code a method decorator named `Listen`, then apply that decorator to `TextInputComponent`. The decorator needs to bind the method it modifies to the second argument of `addEventListener`. Method decorators are just functions, so you'll pass the event name as the first argument and the element that `addEventListener` should be called on as the second argument. This second argument will be optional, as the method decorator can by default call `this.addEventListener` if the end-user doesn't specify a second argument.

To get started, make a new directory in `packages/common/src` named `event`.

```
1 mkdir packages/common/src/event
```

Create a new file named `listen.ts` in `packages/common/src/event`.

```
touch packages/common/src/event/listen.ts
```

Define the Listen decorator

In this file, you'll develop the `Listen` decorator, then export the `function` from the `@in/common` package for use in other packages of the monorepo. First mock the `function`, being sure to make it exportable. The first argument is named `eventName`, type defined as a `string`. This argument will be passed to `addEventListener`. The second argument named `selector` is optional, denoted with `?` in TypeScript and also typed as `string`. This `string` will be passed to `querySelector`.

```
export function Listen(eventName: string, selector?: stri\
ng) {}
```

`Listen` returns a factory function that can be type defined. Make a new type definition using the `type` keyword and name it `MethodDecoratorFactoryFunction`. This type definition will describe the function returned by any method decorator. This `function` has three arguments: `target` is the instance of the `class` the method is found, `key` is the name of the method as a `string`, and descriptor is a `PropertyDescriptor`, which gets its type definition from `lib.dom.d.ts`. `PropertyDescriptor` is an `Object` that describes the configuration of the property on class Object. What's important to remember about the `PropertyDescriptor` is that it stores a property named `value` that equals the actual method being decorated.

Apply the `MethodDecoratorFactoryFunction type` to the returns type of the `Listen function`, then return a `function` named `decorator` and declare all three arguments. The boilerplate for the `Listen` decorator is below.

```
type MethodDecoratorFactoryFunction = (
  target: any,
  key: string | number,
  descriptor: PropertyDescriptor
) => void;

export function Listen(
  eventName: string,
  selector?: string
): MethodDecoratorFactoryFunction {
  return function decorator(
    target: any,
    key: string | number,
    descriptor: PropertyDescriptor
  ) {};
}
```

In the decorator function, we'll need a few things to achieve the end goal.

- `function` that calls `addEventListener`
- `function` that calls `removeEventListener`
- Store the method set as the `function` callback on `addEventListener` so we can call `removeEventListener` with the same method
- Hook into `connectedCallback` and `disconnectedCallback` to call both `function` that add and remove the listeners
- Call `addEventListener` and `removeEventListener` on `this` unless a `selector` is specified, then query for an element using the selector and attach `addEventListener` and `removeEventListener`

That's a lot to unpack. It's not by virtue of being a method decorator that `Listen` is more complex than `Component`. The problem we have to solve is that much more sophisticated. Let's take it step by step.

In `decorator`, access the `connectedCallback` and `disconnectedCallback` methods from the `target` instance using `Object` destructuring. Assign both to noop functions because we don't want to override the user's declared `connectedCallback` and `disconnectedCallback` methods.

```
const { connectedCallback = () => {}, disconnectedCallbac\
k = () => {} } =
  target;
```

We need a reliable way to access the method over time. In one `function`, we'll need to pass the method that is decorated to `addEventListener` and in another method, the same `function` has to be passed to `removeEventListener`. One way to ensure the method never changes is to assign the method to a `Symbol`, which is guaranteed to be unique. Make a new `const` named `symbolMethod` and set this to a new `Symbol`, passing in the `key` (the name of the method decorated by `Listen`). This is mainly so we can access the method like `this[symbolMethod]` throughout the decorator.

```
const symbolMethod = Symbol(key);
```

Next let's write a `function` named `getContext` that takes the `context` in the first argument and detects if a `selector` is provided in the second argument of `Listen` and if truthy, queries the `root` node of the element, depending on if there's Shadow DOM or not. If `selector` is `undefined`, just return `context`. Ternary operators work out nicely here.

```
function getContext(context) {
  const root = context.shadowRoot ? context.shadowRoot : \
context;
  return selector ? root.querySelector(selector) : contex\
t;
}
```

This `function` allows us to determine if the context is `this` or an element returned by `querySelector` when the user passes in a second argument to `Listen`.

Declare a new `function` named `addListener`. In this `function` make a new `const` named `handler` and set it equal to `this[symbolMethod]` and an anonymous `function` that passes in any `arguments` to a call to `descriptor.value.apply(this, args)`. Lets unpack that. `descriptor.value` references the method decorated by `Listen`, but as we find out in the next line, `handler` is passed to `addEventListener` as the `function` callback. Callbacks like this expect an `Event` to be passed through the event handler. This is what is happening here. This algorithm passes the `Event` to the appropriate method by calling `apply` with `this` as the context, and the `arguments` typically passed to the event handler callback.

After the handler is declared, use the `getContext` function to determine where the event listener should be attached, then call `addEventListener`, passing in `eventName` in the first argument and `handler` in the second.

```
function addListener() {
  const handler = (this[symbolMethod] = (...args) => {
    descriptor.value.apply(this, args);
  });
  getContext(this).addEventListener(eventName, handler);
}
```

Removing the event listener is simpler, but only because we've stored the method on `this[symbolMethod]`. Make a new `function`

named `removeListener`. Inside this `function` use `getContext` again to call `removeEventListener`, passing `eventName` in the first argument and `this[symbolMethod]` in the second argument.

```
function removeListener() {
  getContext(this).removeEventListener(eventName, this[sy\
mbolMethod]);
}
```

Finally, override both the `target` instance `connectedCallback` and `disconnectedCallback` by setting both to new `function` that calls the `connectedCallback` and `disconnectedCallback` we originally deconstructed from the `class`, the user's code declared in the same `class` decorated by `Listen`. `connectedCallbackWrapper` and `disconnectedCallbackWrapper` should also should call the `addListener`, and `removeListener`, respectively.

```
target.connectedCallback = function connectedCallbackWrap\
per() {
  connectedCallback.call(this);
  addListener.call(this);
};

target.disconnectedCallback = function disconnectedCallba\
ckWrapper() {
  disconnectedCallback.call(this);
  removeListener.call(this);
};
```

When you are finished, the `Listen` method decorator should look like the example below.

```
type MethodDecoratorFactoryFunction = (
  target: any,
  key: string | number,
  descriptor: PropertyDescriptor
) => void;

export function Listen(
  eventName: string,
  selector?: string
): MethodDecoratorFactoryFunction {
  return function decorator(
    target: any,
    key: string | number,
    descriptor: PropertyDescriptor
  ) {
    const { connectedCallback = () => {}, disconnectedCal\
lback = () => {} } =
      target;
    const symbolMethod = Symbol(key);

    function getContext(context) {
      const root = context.shadowRoot ? context.shadowRoo\
t : context;
      return selector ? root.querySelector(selector) : co\
ntext;
    }

    function addListener() {
      const handler = (this[symbolMethod] = (...args) => {
        descriptor.value.apply(this, args);
      });
      getContext(this).addEventListener(eventName, handle\
r);
    }

```

```
36      function removeListener() {
37        getContext(this).removeEventListener(eventName, thi\
38  s[symbolMethod]);
39      }
40
41      target.connectedCallback = function connectedCallback\
42  Wrapper() {
43        connectedCallback.call(this);
44        addListener.call(this);
45      };
46
47      target.disconnectedCallback = function disconnectedCa\
48  llbackWrapper() {
49        disconnectedCallback.call(this);
50        removeListener.call(this);
51      };
52    };
53  }
```

To use the `Listen` decorator in the context of `TextInputComponent`, you still need to properly export the `function` from the `@in/common` package.

Make a new file named `index.ts` at `packages/common/src/event`.

```
1  touch packages/common/src/event/index.ts
```

Export `Listen` from `packages/common/src/event/index.ts`.

```
1  export { Listen } from "./listen";
```

Export `Listen` from `packages/common/index.ts`.

```
export { Component } from "./src/decorator";
export type { ElementMeta } from "./src/decorator";
export {
  attachShadow,
  attachStyle,
  attachTemplate,
  html,
  css,
} from "./src/template";
export { Listen } from "./src/event";
```

Build the `@in/common` package.

```
lerna exec --scope @in/common -- yarn build
```

Refactor TextInputComponent

To start refactoring `TextInputComponent`, start up Storybook again with `yarn storybook`. If you already have Storybook running you may need to restart it now the `@in/common` package has been updated.

Open `packages/component/src/input/TextInput.ts` to start refactoring `TextInputComponent`.

In this `class` we want to integrate the new `Listen` method decorator. In `connectedCallback`, we set up three event listeners in prior chapters to handle `blur`, `change`, and `keyup` events on the `HTMLInputElement` in this component's template. Before, we accessed this `HTMLInputElement` with a getter on the `class` named `$input`, although now, we have `Listen` which takes a `string` that is passed to `querySelector` to select the `HTMLInputElement`.

There is another problem to overcome. If we apply the `Listen` decorator to `onValidate`, the first argument passed to this method

will now be the `Event` because that's how we set up the logic in the decorator, although `onValidate` takes a `boolean` right now. This `boolean` is ultimately passed to the reusable `validate function`, so it's possible we could deprecate the first argument of `onValidate` and rely only on `validate` instead. This `boolean` is used to toggle whether validate messages and error styling is displayed for the end-user.

Import `Listen` from the `@in/common` package.

```
import { Component, attachShadow, Listen, html, css } fro\
m "@in/common";
```

Deprecate the event listeners one by one. Let's start with the call to `onValidate` that happens when `onblur` is triggered on the `HTMLInputElement`.

Remove the following lines from the `connectedCallback`.

```
this.$input.onblur = () => {
  this.onValidate(true);
};
```

Change the call to `this.onValidate(false)` at the end of `connectedCallback` to instead call `validate` directly.

```
validate(this, false);
```

Find the `onValidate` method and remove the first argument. We don't need to access anything on the `Event` so it's ok to leave the argument blank. This method will only be called onblur so change the call to `validate` to always set `true` for the second argument.

```
validate(this, true);
```

Add the `Listen` decorator to the `onValidate` method. The first argument of `Listen` should be set to `blur`, while the second argument needs to be a string that can query for the `HTMLInputElement`. Since there is only one `HTMLInputElement`, the string can be set to `input`.

```
@Listen('blur', 'input')
onValidate() {
  validate(this, true);
}
```

Navigate to the Form story in Storybook's UI and blur each instance of `TextInputComponent`. Notice the functionality remains the same. The method decorator works!

Find the remaining two event listeners in `connectedCallback`.

```
this.$input.onchange = () => {
  this.onChange();
};
this.$input.onkeyup = () => {
  this.onChange();
};
```

Both of these event listeners call the `onChange` method, but listen for different events: `change` and `keyup`. Remove these two event listeners and apply the `Listen` decorator to the `onChange` method instead, stacking two calls to `Listen` atop `onChange`, one for each `Event`.

```
  @Listen('change', 'input')
  @Listen('keyup', 'input')
  onChange() {
    this.shadowRoot.querySelector('.message').innerHTML =\
 '';
    this.$input.classList.remove('error');
    this.$input.removeAttribute('aria-invalid');
    this.internals.setFormValue(this.value, this.value);
  }
```

That's it for the refactor! Run the tests to ensure nothing is broken.

```
yarn test:e2e
yarn test:snapshot:ci
```

Summary

In this chapter we streamlined Web Component development by coding two TypeScript decorators and several reusable functions.

- `Component` handles custom element registration, template and styling.
- `Listen` creates and destroys event listeners, binding `class` methods to the callback of `addEventListener`
- `attachShadow` handles appending a `ShadowRoot` to any custom element that can accept `ShadowDOM`
- `attachStyle` handles CSS for any custom element that can't encapsulate styles with `ShadowDOM`
- `attachTemplate` appends a HTML template to a custom element that can't use `ShadowDOM`
- `html` and `css` enable syntax highlighting in a user's IDE

We exported these `function` from a new package called `@in/common`, then refactored the three components from Part One to leverage this new micro-library. By learning how to use TypeScript decorators to provide a similar interface as a major JavaScript framework like Angular, we wrangled the primary use cases for custom element development. One feature we didn't tackle was data binding, but even that could be added to the micro-library with a minimal hit to performance. Right now the micro-library is sitting pretty. Without `Listen`, `@in/common` can optimize down to 597 bytes gzipped (Closure Compiler Simple Optimizations). By importing `Listen` into a project, the bundle size for the micro-library becomes 903 bytes gzipped (Closure Compiler Simple Optimizations). Not too shabby for the interface we gain out of `@in/common`.

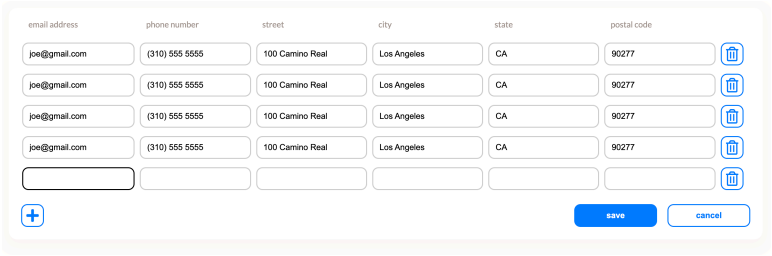
If you wanted more features out of a library you could adopt one of the many open source projects out there. There are many people out there who don't want to "reinvent the wheel" and that is a perfectly valid choice. We focused on this functionality so we can have the best of both worlds: an intuitive and scaffoldable interface for development that compiles down with a minimal performance hit for the end-user. Hopefully by the end of this chapter you understand how some Web Component libraries are coded and even be able to help fix bugs in an open source project or write your own library. Whatever you decide for your next Web Components project: to adopt a library or code your own, it's considered a best practice to use data to guide your decision making process. One thing you've proven in this chapter is that micro-libraries are a viable alternative to JavaScript libraries, often providing a similar solution with much less code.

Compound Custom Elements

Chapter Five

Component Communication

Thus far, we've developed custom elements in isolation. Aside from the `CardComponent` being able to dynamically accept any child element using HTML template slots, we haven't looked at any way custom elements can work seamlessly together. In this chapter, you'll focus exactly on that by building an interactive table component. You'll use browser specifications that enable each component of the table to interact with each other.



Take a moment to review Figma[62]. Users can edit the content of each cell in the table, create and delete table rows, and cancel editing; resetting the value of the table to a state before it was edited. Not only does the table component have to handle CRUD (Create, Read, Update, Delete), the component will also have to have at least

[62]https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=725%3A44

three visible states: read-only, editable, and add row. This means we not only have to provide some sort of state mechanism for `HTMLTableElement`, but we also have to manipulate DOM elements based on changes to table state and react accordingly to different event-based patterns we'll discover while developing the table.

`TableCardComponent` has three modes: read-only, editable, and add row. In read-only, table cells are displayed as text. When the user presses "edit", the table cells transform into editable text fields and focus is placed on the first cell. If the user clicks the "add" button in any mode, the table enters edit mode if needed, and a row of blank text inputs appear for the user to enter a row of information. When the user presses "save", the current state of the table is saved. If the user presses "cancel", the table returns to read-only mode with the state from before the user started editing.

TableComponent

In this exercise, you'll stick with best practices when architecting the components that comprise the interactive experience. You'll use a strict separation of concerns between the card that displays the entire layout: a table and footer with interactive buttons for adding a row, initiating edit mode, cancelling or saving the current edit; the table itself, table rows, and table cells. You'll create several different components that are only concerned with their discrete functionality.

To maintain a separation of concerns, you'll need mechanisms that allow instances of each component to communicate with one another. A robust event-based pattern helps in this regard. The buttons in `TableCardComponent` should dispatch an event that triggers some logic in `TableComponent` to add a row (`TrComponent`). `TrComponent` will control some aspect of how `TdComponent` are displayed.

The three methods for component communication you'll explore

in this chapter are passing data through HTML attributes, dispatching `CustomEvent` to HTML elements directly, and utilizing `BroadcastChannel` for dispatching events where elements that should receive them are not directly accessible.

HTML Attributes

Setting HTML attributes of a custom element are one way two elements can communicate. When a custom element is easily queried, it's relatively simple to set the value of an attribute using `setAttribute`. This method is alright for simple data types, although anything passed through HTML attributes is typically treated as a string. If you wanted to pass a number through a HTML attribute, it gets converted to a string, so you have to coerce it back to a number. This behavior is validated in MDN documentation[63]. Complex data types are not out of the question. As long as you can coerce the data into a string, it can pass through an attribute, although it should be warned, passing PII (Personal Identifiable Information) through HTML attributes is not recommended. Anything passed to a HTML attribute is easily statically analyzable in the DOM by scrapers.

Even though passing data through HTML attributes is rather limiting, it can be extremely useful and convenient. Custom elements provide an API for listening for attribute changes (`observedAttributes`) and doing something when attributes change (`attributeChangedCallback`). When a custom element is reliably a direct descendant or sibling to another custom element, it can be easily queried and natively react to attribute changes.

[63] https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute

CustomEvent

When you have to pass complex data types from one element to another and the element receiving the data is easily selectable via a query, `CustomEvent` is an attractive solution. `CustomEvent` can include a property named `details` that can equal an `Object`, `Array`, `FormData`, or whatever other complex data type you can throw at it. `CustomEvent` can behave like other DOM events (`MouseEvent`), bubbling up the DOM tree, making `CustomEvent` ideal for dispatching events to parent custom elements.

BroadcastChannel

Sometimes elements are not easily accessible either by position in the DOM tree or selectable via a query. `BroadcastChannel` provides a method to dispatch events on a discrete channel that elements can listen to and do something whenever a message is broadcast to the channel. `BroadcastChannel` can dispatch events across DOM and instances of DOM like an embedded iframe or another browser tab, while also overcoming limitations of `CustomEvent` that merely bubbles events up DOM. For some, `BroadcastChannel` may be overkill for simple event dispatching between two elements, but using `BroadcastChannel` prevents attribute drilling, which may be necessary to communicate the selector used to query for another custom element and dispatch a `CustomEvent` or use `setAttribute`. `BroadcastChannel` promotes a loose coupling of user interfaces by not having to pass identifiers around to select another element.

In this chapter, you'll use HTML Attributes, `CustomEvent`, and `BroadcastChannel` while developing the interactive table. The table consists of several different elements that should work cohesively to provide a rich user experience. Throughout this exercise, we'll call the table a "compound component". This isn't an overly technical term, but it fits the scenario: a group of components contained by one another.

What you will build

- `TableCardComponent`, autonomous custom element that displays a specialized `CardComponent` with an editable table and footer with interactive buttons
- `TableComponent`, customized built-in element that extends `HTMLTableElement`, an editable table that displays read-only cells or editable cells
- `TrComponent`, customized built-in element that extends `HTMLTableRowElement` and holds state for a table row
- `TdComponent`, customized built-in element that extends `HTMLTableCellElement` and handles display of a read-only or editable table cell

What you will learn

You'll explore the following concepts:

- How to dispatch and listen for `CustomEvent`
- The limitations of event bubbling and Shadow DOM
- How to use `BroadcastChannel` to dispatch events across DOM
- How to pass data through custom element HTML attributes
- How to handle communication with deeply nested custom elements
- How to work with a separation of concerns in complex custom elements

Acceptance criteria

- `TableCardComponent` displays a specialized `CardComponent` with an editable table and footer with interactive buttons

- `TableCardComponent` uses `BroadcastChannel` to dispatch events to other custom elements
- `TableComponent` displays a `HTMLTableHeaderElement` with read only column names
- `TableComponent` handles rendering of `HTMLTableRowElement`
- `TableComponent` handles validation of `HTMLTableRowElement`
- `TableComponent` handles state of table data, provides method for creation and deletion of `HTMLTableRowElement`
- `TableComponent` handles state of table modes, i.e. read only or editable
- `TrComponent` handles state of a single table row
- `TdComponent` handles the display of a single table cell
- All components appear to follow the design guidelines in Figma

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-5 from where you left off at the end of chapter 4.

```
git checkout -b chapter-5
```

Mocking Table Data in Storybook

For starters we're going to craft a data structure for the table component. The interactive table should render based on a data

object passed to it that includes static data for the columns and rows of the table. We can mock this data in Storybook and pass the data structure to the component via a Story. We'll need both a component and story file to get started, so let's create these files while providing some basic boilerplate in each file.

TableCardComponent

Make a new directory that houses all the files we'll code for the table component.

```
mkdir packages/component/src/table
```

The outermost component necessary to render the interactive table is the `TableCardComponent`. This component renders a `CardComponent` with a `TableComponent` and several buttons in the card footer. Make a new file for `TableCardComponent` named `TableCard.ts` in the same directory you just created.

```
touch packages/component/src/table/TableCard.ts
```

Open `TableCard.ts` in your IDE and begin developing the component. Import the necessary API from `@in/common`. `TableCardComponent` is an autonomous custom element so you'll need `Component` and `attachShadow`. You'll give the autonomous custom element a template and styling, so best to also import `html` and `css`.

```
import { Component, attachShadow, html, css } from "@in/c\
ommon";
```

Next, declare the boilerplate for `TableCardComponent`. Since `TableCardComponent` is an autonomous custom element, you

should have a `class` that extends `HTMLElement`. After ensuring the `class` is properly exported, use the `@Component` decorator to declare the tag name of the custom element: `in-tablecard`. Follow up by giving this new component a blank style and template. In the `constructor` call `attachShadow`, passing in `this`, the context equaling the instance of `TableCardComponent`.

```
@Component({
  selector: "in-tablecard",
  style: css``,
  template: html``,
})
export class TableCardComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

Next, create another file in the same table directory, this time to house the Story. Name the file Table.`stories.js`.

```
touch packages/component/src/table/Table.stories.js
```

In `Table.stories.js`, export an `Object` using `default` per the Storybook Component Story Format, giving the story a `title` and `component`.

```
export default {
  title: "Components/Table",
  component: "in-tablecard",
};
```

Import the necessary APIs to render the story template. You'll need the `html function` from lit-html for Storybook to correctly

interpolate `string` passed to the story template. Next, import `TableCardComponent` you created earlier in `TableCard.ts`.

```
import { html } from "lit-html";
import { TableCardComponent } from "./TableCard";
```

In `Table.stories.js`, export `TableCardComponent` to ensure Storybook can properly build the story file. This is only currently necessary for Storybook's Webpack configuration to properly build the project and can possibly be omitted at a later date.

```
export { TableCardComponent };
```

Create a new `const` named `Template` and make it equal to a `function` that returns `html`. Pass `html` a `string` in the `function` first argument using shorthand syntax, a `string` that includes the `in-tablecard` element. Afterward, create another `const` named `Primary`, making it equal to a call to the `Template.bind` method, passing in an empty `Object` for now to this method. Later you'll populate the `bind` method with any properties that should be passed to the template. Finally, instantiate the `Primary` story's `args` with an empty `Object`. This is where you'll provide mock data for the `Primary` story.

```
const Template = () => {
  return html`<in-tablecard></in-tablecard>`;
};

export const Primary = Template.bind({});
Primary.args = {};
```

`TableCardComponent` is merely a container that should include another component, `TableComponent`, we have yet to develop. `TableComponent` will render the interactive table, while

`TableCardComponent` is mainly responsible for providing a template that displays an instance of `TableComponent` along with some `ButtonComponent` in the `CardComponent` footer.

TableComponent and ColumnData

Make a new file that will include the majority of code for this chapter: `Table.ts`.

```
touch packages/component/src/table/Table.ts
```

Pretty soon you'll need to provide some mock data for the table. That mock data includes an `Array` of `Object` that describe the table columns. Since `TableComponent` is the main receptacle for this data, let's make a new type interface that describes a single `Column`. Export a new `interface` named `Column`, which resembles an `Object`, then declare each property needed on the `Column interface`.

```
export interface Column {
  property: string;
  label: string;
  span?: number;
  align: "left" | "center" | "right" | "justify";
  index: number;
}
```

Each column will have a label displayed on the top of the column that provides contextual information for the end-user. Make a new `property` on the `interface` named `label` and type define it as `string`. `HTMLTableElement` columns take a property named `colSpan` that determine the number of columns the column can span across the table, so let's provide a means for engineers to set the `colSpan` via this interface. Name a new property `span` and make it optional using `?` and type defined as a `number`. Since `Column` will be stored in

an `Array` it could be useful to store the index of the column on this interface so we don't have to depend on `Array` methods to figure out the index. Set another property named `index` to `number`. Table columns may be styled in several ways, but most often designers want to align text a certain way. Provide another property named `align` that can be set to various ways that will eventually style the column.

Finally and most crucial for rendering the table with various data sets, there should be a way to map each column to a property on the data for each row. Give the `interface` a new property named `property` and type define it as `string`. In the `TableComponent` implementation, `property` will be used to map each cell in a row to the column it should display in.

Tables rarely display a single column, so engineers will need a way to declare multiple columns. This can be easily done by declaring an `Array` of `Column`. Export another `type` named `ColumnData` and make it equal to `Column[]`.

```
export type ColumnData = Column[];
```

In the same file, import all the necessary API from `@in/common` to develop a customized built-in element.

```
import { Component, attachTemplate, attachStyle, html, cs\
s } from "@in/common";
```

Declare a new `class` named `TableComponent` that extends `HTMLTableElement`. This will allow engineers to use `TableComponent` like any other `HTMLTableElement`. In the `constructor`, call `attachTemplate` and `attachStyle`, passing in the context of the instance of `TableComponent`: `this` into both functions imported from `@in/common`. Declare a new `@Component` decorator and pass in an `Object` with the `selector` property set to the tag name of the new component: `in-table`. Set the `custom` property to an `Object`

that gets passed to the `CustomElementRegistry.define` method for customized built-in elements. Give the decorator a `style` and `template` properties, while using the `css` and `html function` imported from `@in/common`. You can leave `style` blank for now, but give the `template` the basic template of a `HTMLTableElement`: two HTML elements that include the template for the table header: `<thead>` and body `<tbody>`.

```
@Component({
  selector: "in-table",
  custom: { extends: "table" },
  style: css``,
  template: html`
    <thead></thead>
    <tbody></tbody>
  `,
})
export class TableComponent extends HTMLTableElement {
  constructor() {
    super();
    attachTemplate(this);
    attachStyle(this);
  }
}
```

When you are finished declaring the boilerplate for `TableComponent`, import both `ColumnData` and `TableComponent` into `TableCard.stories.js`

```
import { TableComponent, ColumnData } from "./Table";
```

Export both `TableComponent` and `ColumnData` in the same file to satisfy the Webpack configuration for Storybook.

```
export { TableCardComponent, TableComponent, ColumnData };
```

Mock Data

`TableComponent` should render any `Array` of `Object` as rows and accept configuration for columns in the form of `ColumnData`. Let's look first at the data for each row. In the application this component is being developed, the table will display a contact list. Each contact has an email address, phone number, and physical address. Usually a physical address is split into many form fields, so the street, city, region, and postal code are separate properties on the same `Object`.

Create a new `const` in `Table.stories.js` named `rowData` and make it equal to an `Array` of `Object`. Each `Object` should have fix properties: `email`, `phoneNumber`, `street`, `city`, `region`, and `postalCode`. Populate the `Array` with a few `Object` so the table can render multiple rows.

```
const rowData = [
  {
    email: "joe@gmail.com",
    phoneNumber: "(310) 555 5555",
    street: "100 Camino Real",
    city: "Los Angeles",
    region: "CA",
    postalCode: "90277",
  },
  {
    email: "joe@gmail.com",
    phoneNumber: "(310) 555 5555",
    street: "100 Camino Real",
    city: "Los Angeles",
    region: "CA",
    postalCode: "90277",
```

```
  },
  {
    email: "joe@gmail.com",
    phoneNumber: "(310) 555 5555",
    street: "100 Camino Real",
    city: "Los Angeles",
    region: "CA",
    postalCode: "90277",
  },
  {
    email: "joe@gmail.com",
    phoneNumber: "(310) 555 5555",
    street: "100 Camino Real",
    city: "Los Angeles",
    region: "CA",
    postalCode: "90277",
  },
];
```

Next, mock the `ColumnData` we type defined earlier. Make a new `const` named `columnData` and make it equal to an `Array`, define the type as `ColumnData` which should already be imported from `Table.ts`. Add six `Column` to the `Array`, one for each property in an `Object` found in `rowData`. For example, the first `Column` should have a property named `property` that is equal to `email`. The `label` property should be set to `email address` so the column is properly labeled. A peculiar aspect to `ColumnData` is that it is unordered, simulating the possible unordered return from a server. Each `Column` provides its index in the `Array` via the `index` property. See the example below for a complete `ColumnData`.

```
const columnData: ColumnData = [
  {
    property: "email",
    label: "email address",
    index: 0,
    align: "left",
  },
  {
    property: "postalCode",
    label: "postal code",
    index: 5,
    align: "left",
  },
  {
    property: "region",
    label: "state",
    index: 4,
    align: "left",
  },
  {
    property: "city",
    label: "city",
    index: 3,
    align: "left",
  },
  {
    property: "street",
    label: "street",
    index: 2,
    align: "left",
  },
  {
    property: "phoneNumber",
    label: "phone number",
    index: 1,
```

```
    align: "left",
  },
];
```

We could keep the column and row data models separate. Although, if the plan is to dispatch an event with both, it would be handy to place both data models on the same `Object`. Create a new `const` named `TableContext` and set two properties, `rowData` and `columnData`.

```
const TableContext = {
  rowData,
  columnData,
};
```

Eventually, we'll need to pass `TableContext` to `TableComponent` but before we even do that we should set up the story template. Add a property named `context` on `Primary.args` and set the value of this new property with the `TableContext` you just created.

```
Primary.args = {
  context: TableContext,
};
```

Pass the `context` to the `Template function` using Object destructuring.

```
const Template = ({ context }) => {
```

Since `TableComponent` is deeply nested in the layout and `TableContext` is a complex `Object`, we probably don't want to set an attribute of `in-tablecard` with this new data model we just mocked. Instead, in the next section of this chapter, we'll explore `BroadcastChannel` in detail and use it to send

`TableContext` to `TableComponent`. For now, all you need to know is `BroadcastChannel` requires a name for the channel where all communication between elements occur. Add a new property to `Primary.args` named `channelName` and set it to a `string` with the value `table:one`.

```
Primary.args = {
  channelName: "table:one",
  context: TableContext,
};
```

Just like `context`, pass `channelName` through the `Template function` using Object destructuring.

```
const Template = ({ channelName, context }) => {
```

Set an attribute on `in-tablecard` named `channel` to `channelName`. Since we're using Lit-html function in this context we can use String interpolation in a template literal to set the attribute in the template.

```
return html`<in-tablecard channel="${channelName}"></in-t\
ablecard>`;
```

If some of the above boilerplate seems murky, thats alright. We have yet to do something meaningful with the `channelName` and `context` passed through `Template`. So far you've used one method of component communication: setting a HTML attribute to provide the channel name to `TableCardComponent` but we have yet to react to the attribute changes in any meaningful way. `context` is similarly hanging around without any implementation, although you've set up boilerplate for both `TableCardComponent` and `TableComponent` and provided mock data to the `Primary` story in Storybook. In the next section of this chapter, you'll explore the three ways of component communication in-depth and work closely with the components to provide a rich, interactive user experience.

Coding Compound Components

In this section, you'll explore two ways for custom elements to communicate with one another: setting HTML attributes and sending messages over `BroadcastChannel`. First, you'll use HTML attributes to provide the channel name to both `TableCardComponent` and `TableComponent`. Currently the channel name is set on an attribute of an instance of `TableCardComponent`. You'll need to update the template of `TableCardComponent` to include an instance of `TableComponent` and pass the name of the channel using attribute drilling to `TableComponent`.

`TableComponent` is ultimately the component that requires the mock data to render columns and rows. You'll use the channel name passed through an attribute on `TableComponent` to set up `BroadcastChannel` internally and subscribe to messages broadcast on the channel. In the story file, you'll broadcast a message that contains the mock data to `TableComponent`.

Attribute Drilling

Custom elements have robust ways to listen for changes on HTML attributes (`observedAttributes`) and react to those changes (`attributeChangedCallback`). Setting HTML attributes on a custom element is a perfectly viable way to communicate with the element, provided the element is queriable. Since `TableComponent` is a direct child of `TableCardComponent`, you can select the element with `querySelector` and use `setAttribute` to change the attribute.

To get started, update the template of `TableCardComponent` to include a `CardComponent`. `CardComponent` has several HTML template slots you can set content.

```
@Component({
  selector: 'in-tablecard',
  style: css``,
  template: html`
    <in-card>
      <table is="in-table" slot="content"></table>
      <div class="table-footer" slot="footer"></div>
    </in-card>
  `,
})
```

`TableComponent` is a customized built-in element, so you can use it like any other `HTMLTableElement`. Add a `HTMLTableElement` to the inner content of the `CardComponent` and set the `slot` attribute to `content` so the table displays in the main content area of the card. Next, add a `HTMLDivElement`, give it a CSS class name set to `table-footer`, and set the `slot` attribute to `footer`. You'll use the class name later to style the footer.

Import `CardComponent` into `Table.stories.js` so you can use the component in the context of the story.

```
import { CardComponent } from "./../card/Card";
```

Additionally, export `CardComponent` to satisfy Storybook's Webpack compilation.

```
export { CardComponent, TableCardComponent, TableComponen\
t, ColumnData };
```

Back in `TableCard.ts`, add the `observedAttributes` method as a static getter on `TableCardComponent` and return an `Array` with the `String` `channel` set as the only item in the `Array`. This enables the custom element to listen for changes on a HTML attribute with the name `channel`.

```
static get observedAttributes() {
  return ['channel'];
}
```

To pass the value of the `TableCardComponent` `channel` HTML attribute to `TableComponent` you'll first need to select the instance of `TableComponent` using `querySelector`. An easy way to do this is via a getter on `TableCardComponent` named `$table`. With this method you can keep `TableComponent` strictly typed by setting the return type of the getter method to `HTMLTableElement`. Finally, return `this.shadowRoot.querySelector('table')` in the getter method. You need to query for the `HTMLTableElement` on the `shadowRoot` because `TableCardComponent` is an autonomous custom element employing Shadow DOM.

```
get $table(): HTMLTableElement {
  return this.shadowRoot.querySelector('table');
}
```

In the `attributeChangedCallback` you'll reference `this.$table` to pass the value of the `channel` attribute from `TableCardComponent` to `TableComponent`. You'll also set up a new instance of `BroadcastChannel` in the `attributeChangedCallback` and store the instance on a new property of `TableCardComponent` we'll name `channel`. Set the type definition of the `channel` prior to the `constructor` in `TableCardComponent`.

```
private channel: BroadcastChannel;
```

Set up the `attributeChangedCallback` method to react to changes to the value of the `channel` HTML attribute. Just like in previous chapters, use the `name` and `next` arguments of the "attributeChangedCallbackmethod and aswitchstatement. In thechannel, switchcase, set up anew BroadcastChanneland

pass in the value of the `channel` attribute to the first argument using `next`. `BroadcastChannel` requires a channel name, at the very least. We'll leave the `channel` here for later, in case we need to react to messages on the channel in `TableCardComponent`. Next, reference the `TableComponent` with `this.$table` and use `setAttribute` to pass the `channel` attribute value to `TableComponent`.

```
attributeChangedCallback(name, prev, next) {
  switch (name) {
    case 'channel':
      this.channel = new BroadcastChannel(next);
      this.$table.setAttribute('channel', next);
      break;
  }
}
```

In `TableComponent`, we still need a way to listen for changes to the `channel` attribute that you just set via the `attributeChangedCallback` in `TableCardComponent`. Set up a similar `observedAttributes` static getter in `Table.ts` to listen for changes on the `channel` attribute.

```
static get observedAttributes() {
  return ['channel'];
}
```

Just like in `TableCardComponent`, you'll need to create an instance of `BroadcastChannel` in `TableComponent` to listen for messages on the channel. Setup a property on the `TableComponent` `class` type defined as `BroadcastChannel`.

```
private channel: BroadcastChannel;
```

The `attributeChangedCallback` on `TableComponent` should be similar to `TableCardComponent`. Listen for changes on the `channel` attribute and set up an instance of `BroadcastChannel`. This time, call `onmessage`, a method available on the instance of `BroadcastChannel` that lets you bind a `function` and listen for messages. Set the value of `this.channel.onmessage` to a class method we haven't made yet: `this.onMessage`, making sure to call the method with `bind` otherwise the context inside `onMessage` will be the `BroadcastChannel` instead of `TableComponent`.

```
attributeChangedCallback(name, prev, next) {
  switch (name) {
    case 'channel':
      this.channel = new BroadcastChannel(next);
      this.channel.onmessage = this.onMessage.bind(this);
      break;
  }
}
```

Add the `onMessage` method on `TableComponent`. `onMessage` behaves like other event listeners. Whenever a message is broadcast on the channel, an event passes through the first argument of `onMessage` since it's bound to `this.channel.onmessage`. Set up a `console.log` temporarily to easily inspect messages on `BroadcastChannel`.

```
onMessage(ev) {
  console.log(ev.data);
}
```

BroadcastChannel

So far you've set up `TableComponent` to listen for messages on `BroadcastChannel` but nothing is actually dispatching messages

on the channel. Earlier in `Table.stories.js` you established the `channelName` that was set on the `channel` HTML attribute of `TableCardComponent`.

In `Template`, use the `channelName` to instantiate yet another instance of `BroadcastChannel`, only this time dispatch a message on the channel using the `postMessage` method, passing this method an `Object` containing two properties: `type` and `detail`. This `Object` can contain any custom properties you desire. Since many different messages are going to be broadcast on the same channel, it's a good idea to assign `type` to distinguish one kind of message from another. Set `type` to `data` because this message contains the data model for the table. `detail` should contain the message you want to transmit on the channel.

Earlier, you set the column and row data on an `Object` named `TableContext` and passed that to the template with the `context` argument. Set `detail` to the `context`.

```
const channel = new BroadcastChannel(channelName);
channel.postMessage({
  type: "data",
  detail: context,
});
```

Now that you're dispatching a message on the `BroadcastChannel` and `TableComponent` is listening via the `this.channel.onmessage` callback, you should inspect the console log. Open up Dev Tools and notice nothing is logged. This is because Storybook, more accurately LitElement `html`, requires at least one tick of the JavaScript engine to compile the template. Wrap the call to `postMessage` in a `setTimeout` to address this behavior.

```
const channel = new BroadcastChannel(channelName);
setTimeout(() => {
  channel.postMessage({
    type: "data",
    detail: context,
  });
}, 0);
```

You should now see the console log in Dev Tools.

Rendering Columns and Rows

Messages are flowing from the story to `TableComponent` but we don't have a method to react to the message except for a `console.log` at this point. The message broadcast to `TableComponent` contains a data model for displaying columns and rows in the table. To react to this data model, replace the `console.log` in `onMessage` with a call to a yet unrealized method: `onTableData`, making sure to pass `ev.data.detail` to the new method.

```
onMessage(ev) {
  switch (ev.data.type) {
    case 'data':
      this.onTableData(ev.data.detail);
      break;
  }
}
```

Declare the `onTableData` method and give it an argument. We'll call this argument `next` for lack of a better name. Rendering the header and body of the table are completely different algorithms, so split up the rendering of these table sections into two methods:

`renderHeader` and `renderRows`. Pass each their respective property on the data model: `next.columnData` and `next.rowData`.

```
onTableData(next) {
  this.renderHeader(next.columnData);
  this.renderRows(next.rowData);
}
```

Rendering Column Data

We need to set up a couple of items before diving into the `renderHeader` method. We'll store the columnar data on a new property declared on `TableComponent`: `columnData`.

```
private columnData: ColumnData;
```

We'll need a way to reference the `TableHeadElement` and one way to do that without multiple calls to `querySelector` is to provide a getter on the `TableComponent class`.

```
get $head() {
  return this.querySelector('thead');
}
```

Declare the `renderHeader` method and pass in the `ColumnData` through the first argument.

```
renderHeader(cols: ColumnData) {

}
```

If you inspect the `ColumnData` (through `console.log` or breakpoint), you'll observe the columns are out of order in the `Array`,

but each `Column` contains a property for `index`. `TableComponent` should display the columns indexed properly from left to right. Use `Array.prototype.sort` to order the columns and set the ordered `Array` on `this.columnData`.

```
this.columnData = cols.sort((a, b) => a.index - b.index);
```

Next is a matter of generating the appropriate DOM elements in the `TableHeaderElement`. A `TableRowElement` should be created that will display a row of `TableHeaderElement`, each responsible for displaying the column label in the table header.

```
const tr = document.createElement("tr");
```

Loop through the `cols` data model using `Array.prototype.forEach`. On each iteration of the loop, make a new `TableHeaderElement`, set the `innerText` of the `TableHeaderElement` with the `label` property on `Column`, then append the `TableHeaderElement` to `TableRowElement`.

```
cols.forEach((colData) => {
  const th = document.createElement("th");
  th.innerText = colData.label;
  tr.appendChild(th);
});
```

Finally, reset the content of the `TableHeadElement` every time `renderHeader` is called. This will allow re-rendering the `TableHeadElement` with different columns. On the last line of `renderHeader`, append the `TableRowElement` to `TableHeadElement`.

```
this.$head.innerHTML = "";
this.$head.appendChild(tr);
```

When you are finished with the `renderHeader` method, it should appear as below.

```
renderHeader(cols: ColumnData) {
  this.columnData = cols.sort((a, b) => a.index - b.index\
);
  const tr = document.createElement('tr');

  cols.forEach((colData) => {
    const th = document.createElement('th');
    th.innerText = colData.label;
    tr.appendChild(th);
  });

  this.$head.innerHTML = '';
  this.$head.appendChild(tr);
}
```

Rendering Row Data

The algorithm for rendering rows is similar to `renderHeader` that renders columns, although this time we should need to store the rows on the `class`. The reason we have to store `columnData` on the `class` should become apparent in the implementation of `renderRows`, which relies on `columnData` to render each `TableDataCellElement`.

`columnData` is an ordered `Array` which is necessary to display the `TableDataCellElement` in the correct order.

To get started, you'll need a way to reference the `TableBodyElement` in the `renderRows` method. Rows will be appended to this element.

Make another getter named `$body` on `TableComponent` and make it return the `tbody`.

```
get $body() {
  return this.querySelector('tbody');
}
```

Declare a new method named `renderRows` and pass in the `rows` in the first argument. You can type define `rows` as `any[]` for the time being, because literally any data model could be displayed in the row.

```
renderRows(rows: any[]) {

}
```

Reset the content of the `TableBodyElement` so this method can be used to render the rows more than once, allowing users of this component to replace the data displayed in the table.

```
this.$body.innerHTML = "";
```

Using `Array.prototype.forEach`, loop over each row and generate an empty `TableRowElement`.

```
rows.forEach((rowData) => {
  const tr = document.createElement("tr");
});
```

Inside the loop, on every iteration, place a nested loop using `Array.prototype.forEach` yet again to loop over the `Array` of `Column` in `this.columnData`. On each iteration of the nested loop, generate a new `TableDataCellElement`, set the `innerText` to the intended content of the cell. Earlier, we had mapped a known

property on the data model to the `Column`. Use the `colData.property` now to reference the data for the `TableDataCellElement`. Finally, append the `TableDataCellElement` to the `TableRowElement`.

```
this.columnData.forEach((colData) => {
  const td = document.createElement("td");
  td.innerText = rowData[colData.property];
  tr.appendChild(td);
});
```

After the nested `forEach`, make sure you append the `TableRowElement` to the `TableBodyElement` to complete the iteration of the loop that iterates over each row.

```
this.$body.appendChild(tr);
```

When you are finished coding `renderRows` it should appear like the example below:

```
renderRows(rows: any[]) {
  this.$body.innerHTML = '';
  rows.forEach((rowData) => {
    const tr = document.createElement('tr');
    this.columnData.forEach((colData) => {
      const td = document.createElement('td');
      td.innerText = rowData[colData.property];
      tr.appendChild(td);
    });
    this.$body.appendChild(tr);
  });
}
```

After completing `renderRows` and `renderHeader`, you could observe the row and column data being displayed in Storybook. However, the content is not styled and appears to be cut off in the `CardComponent`, so let's fix that.

Styling TableComponent

One of the problems with coding compound components is finding architectural patterns that satisfy many different combinations of components, as is the case with `CardComponent`, which is currently styled in a way where the component's width is restricted to `320px`. In earlier chapters, this styling was to facilitiate the display of a marketing card and login form; however, this styling is now prohibiting us from using `CardComponent` to display much larger user experiences like `TableComponent`.

To fix this problem, remove the `max-width: 320px` style in `CardComponent`.

```
max-width: 320px;
```

In `Card.stories.js`, add this style back as an inline style for now to satisfy the requirements of those stories. Verify in Storybook that the existing card stories retain the expected width.

```
<in-card style="max-width: 320px"></in-card>
```

Moving back to the Table story in Storybook to observe styling updates to `TableComponent`, focus now on `Table.ts`. Earlier, we had set up this customized built-in element with a call to `attachStyle` which should place a `<style>` tag in the `document.head` (or parent component `shadowRoot`) based off the work you did in Chapter Four.

```
attachStyle(this);
```

In `Table.ts`, fill out the `style` property on the `Component` decorator with the appropriate styles. Reference Figma to obtain the font, alignment, and sizing of the `TableComponent` and its children,

`TableRowElement` and `TableDataCellElement`. You may inspect your work in Storybook as you progress through styling `TableComponent`.

First, select the `TableComponent` using an attribute selector `[is="in-table"]` since this component is a customized built-in element. Style the font properties of the table, specifiying `font-family`, `font-size`, `font-weight`, and `color`. Make `TableComponent` always fill the width of its container by specifying `width: 100%`.

```
style: css`
  [is='in-table'] {
    font-family: var(--font-default);
    font-size: var(--font-body-md);
    font-weight: var(--font-weight-default);
    color: var(--color-neutral-500);
    width: 100%;
  }
`
```

Turning our attention to the `TableHeaderElement`, specify the necessary styles to reflect the mockup in Figma. Set the `display` to `table-cell` to normalize the styling for different browsers. Center the label vertically using `vertical-align: middle`, which overrides the default. Set the `box-sizing: border-box` to more predictably size the element using only width, height, padding and border. Give the `TableHeaderElement` the appropriate `margin-top` and `height` and set the `font-weight` to override the browser default.

```
[is="in-table"] th {
  display: table-cell;
  box-sizing: border-box;
  margin-top: var(--margin-xs);
  height: 44px;
  vertical-align: middle;
  font-weight: var(--font-weight-default);
}
```

Figma specifies the height of each `TableRowElement` should be `58px`. To center content vertically, override the browser default with `vertical-align: middle`.

```
[is="in-table"] tr {
  height: 58px;
  vertical-align: middle;
}
```

Each `TableDataCellElement` requires some padding to match the Figma mockup.

```
[is="in-table"] td {
  padding-left: var(--padding-xxs);
  padding-right: var(--padding-xxs);
}
```

When you are finished styling `TableComponent`, the `style` property on the `Component` decorator should appear as below.

```
style: css`
  [is='in-table'] {
    font-family: var(--font-default);
    font-size: var(--font-body-md);
    font-weight: var(--font-weight-default);
    color: var(--color-neutral-500);
    width: 100%;
  }
  [is='in-table'] th {
    display: table-cell;
    box-sizing: border-box;
    margin-top: var(--margin-xs);
    height: 44px;
    vertical-align: middle;
    font-weight: var(--font-weight-default);
  }
  [is='in-table'] tr {
    height: 58px;
    vertical-align: middle;
  }
  [is='in-table'] td {
    padding-left: var(--padding-xxs);
    padding-right: var(--padding-xxs);
  }
`,
```

After taking a brief detour to style `TableComponent`, let's circle back to other functions of the user experience that require more work.

Handling CRUD Events with BroadcastChannel

If we are going to support CRUD (Create, Read, Update and Delete) in `TableComponent`, we need a way to tell `TableComponent` when

to create, update, delete, or read rows. Looking back at Figma, several buttons are visible in the layout that appear to initiate such actions. `Edit` button puts `TableComponent` in a mode that makes each cell editable. In "edit mode", two buttons appear: `Save` and `Cancel`. When the user presses `Save`, `TableComponent` should revert to "read only mode" with any changes saved to each cell. When the user presses `Cancel`, `TableComponent` should revert to "read only mode" with the state prior to the user making any changes. Later, we'll include an `Add` button for creating a new row and icons in each row to support `Delete`, but for now let's focus on the R (`Read`) and U (`Update`) in CRUD.

Add a new `HTMLDivElement` to `TableCardComponent` template, giving it an identifiable CSS class name `save-actions`. In this `HTMLDivElement`, add three `ButtonComponent` we developed in prior chapters using customized built-in elements. The three buttons should have the appropriate labels: `save`, `cancel`, and `edit`. `save` is a primary button, while `cancel` and `edit` are secondary buttons. Initially, the first two buttons (`save` and `cancel`) should be hidden. This can be achieved with the `hidden` HTML attribute.

```
<div class="table-footer" slot="footer">
  <div class="save-actions">
    <button class="primary button-save" is="in-button" hi\
dden>save</button>
    <button class="secondary button-cancel" is="in-button\
" hidden>
      cancel
    </button>
    <button class="secondary button-edit" is="in-button">\
edit</button>
  </div>
</div>
```

Add three getters on `TableCardComponent`, one for each of the buttons you just introduced to the template. If you haven't already,

give each `ButtonComponent` a unique CSS class name so you can query for each element easily using `querySelector`.

```
  get $editButton(): HTMLElement {
    return this.shadowRoot.querySelector('.button-edit');
  }

  get $saveButton(): HTMLElement {
    return this.shadowRoot.querySelector('.button-save');
  }

  get $cancelButton(): HTMLElement {
    return this.shadowRoot.querySelector('.button-cancel'\
);
  }
```

Focusing on the edit button, make a new event listener using the `Listen` decorator from Chapter 4. Import `Listen` from `@in/common` and declare the decorator atop a new method named `editMode`. The event listener should listen for `click` on the element with class name `.button-edit`. In the callback, add the `hidden` HTML attribute to `this.$editButton` and remove the same attribute on `this.$saveButton` and `this.$cancelButton` to hide the edit button and show the cancel and save buttons. Call `postMessage` on `this.channel`, setting the `type` of the message to `edit`. There isn't a payload needed here, so we'll leave `detail` out. When the user clicks the edit button, a new message is broadcast via `BroadcastChannel` that specifies that the mode has changed to "edit mode".

```
@Listen('click', '.button-edit')
editMode() {
  this.$editButton.setAttribute('hidden', 'true');
  this.$saveButton.removeAttribute('hidden');
  this.$cancelButton.removeAttribute('hidden');
  this.channel.postMessage({
    type: 'edit',
  });
}
```

The cancel button should have a similar event listener. Use another instance of the `Listen` decorator to establish a `click` event listener on the element with class name `.button-cancel`. Attach the decorator to a new method named `readOnlyMode`. Show the edit button, while hiding the save and cancel buttons by removing and adding the `hidden` attribute respectively. Dispatch a message on the `BroadcastChannel`, this time setting the `type` to `edit`.

```
@Listen('click', '.button-cancel')
readOnlyMode() {
  this.$editButton.removeAttribute('hidden');
  this.$saveButton.setAttribute('hidden', 'true');
  this.$cancelButton.setAttribute('hidden', 'true');
  this.channel.postMessage({
    type: 'readOnly',
  });
}
```

Declare a third event listener attached to the save button. In a new `save` method, dispatch an event on the `BroadcastChannel` with the `type` property set to `save`. We could hide and show buttons similar to the cancel callback, but will reserve that functionality only for the `readOnlyMode` method that can be called independent of the event listener callback since it is a `class` member of `TableCardComponent`.

```
@Listen('click', '.button-save')
save() {
  this.channel.postMessage({
    type: 'save',
  });
}
```

Ultimately these messages should be received by `TableComponent`, so update that `class` with new `case` in the `switch` statement found in the `onMessage` method. React to each of the types of messages: `edit`, `readOnly`, and `save`. In each `case`, call a new method for each use case.

```
onMessage(ev) {
  switch (ev.data.type) {
    case 'data':
      this.onTableData(ev.data.detail);
      break;
    case 'edit':
      this.onEdit();
      break;
    case 'readOnly':
      this.onReadOnly();
      break;
    case 'save':
      this.onSave();
      break;
  }
}
```

Add the new methods to `TableComponent`.

```
onEdit() {}
onReadOnly() {}
onSave() {}
```

In this section you worked with several components to develop a rich user experience. `TableCardComponent`, `TableComponent`, `CardComponent` and `ButtonComponent` all work together through the use of attribute drilling or dispatching messages on the `BroadcastChannel` event bus. In the next section, you'll focus mainly on `TableComponent` to achieve all the features necessary to make an editable table.

Making the Table Editable

In this section, you'll work on giving `TableComponent` the various modes required by the acceptance criteria: read-only and editable, while also giving the user the ability to add and delete rows. You'll further practice a separation of concerns by developing two new components: `TrComponent` and `TdComponent` that handle the functionality of a row and cell, respectively.

We used attribute drilling and `BroadcastChannel` in the previous section to communicate between components in the same layout. We'll add `CustomEvent` as a method for custom element communication in this section. You'll also learn how to dynamically create new instances of customized built-in elements with `createElement` and give custom elements data with HTML data attributes.

Separation of Concerns

In the last section, you dynamically created `HTMLTableRowElement` and `HTMLTableCellElement` in `TableComponent` in response to receiving a message over `BroadcastChannel`. To further promote a

separation of concerns, let's break out these two elements into new components. Otherwise, logic that should probably be contained by the `HTMLTableCellElement` will be leaking all over `TableComponent`. `TrComponent` and `TdComponent` will be customized built-in elements because we want them to retain the characteristics of the native DOM elements, while enhancing the element with more functionality. `TrComponent` will manage state for the entire row, while `TdComponent` will manage the view of the `HTMLTableCellElement`, whether it's in "edit mode" or "read-only mode", and set the value of the cell.

To start, make a new file in the same directory named `Tr.ts`. This file will contain the component logic for `TrComponent`.

```
touch packages/component/src/table/Tr.ts
```

Open `Tr.ts` in your IDE and begin developing the component by importing the `Component` decorator from `@in/common`.

```
import { Component } from "@in/common";
```

Declare a new `class` named `TrComponent` and have it extend `HTMLTableRowElement`. Make the `constructor` and insert a call to `super()`. Ensure the `class` is exported using the `export` keyword.

```
export class TrComponent extends HTMLTableRowElement {
  constructor() {
    super();
  }
}
```

Use the `Component` decorator to set the custom element tag name as `in-tr`, making sure to also provide the necessary `custom` property needed to register customized built-in elements.

```
@Component({
  selector: 'in-tr',
  custom: { extends: 'tr' },
})
```

In `Table.stories.js` import `TrComponent` from `Tr.ts`.

```
import { TrComponent } from "./Tr";
```

Export `TrComponent` for the Storybook Webpack configuration.

```
export {
  CardComponent,
  TableCardComponent,
  TableComponent,
  TrComponent,
  ColumnData,
};
```

Set up `TrComponent` the same way. Start by making a new file named `Td.ts` in the same directory.

```
touch packages/component/src/table/Td.ts
```

Import all the necessary API from `@in/common`. You'll need to style parts of a custom template in `TdComponent` so be sure to import `attachTemplate` and `attachStyle` in addition to `Component`.

```
import { Component, attachTemplate, attachStyle, html, cs\
s } from "@in/common";
```

Export a class named `TrComponent` that extends `HTMLTableCellElement`. Call `super()` in the `class constructor` as well as `attachTemplate` and `attachStyle`.

```
export class TdComponent extends HTMLTableCellElement {
  constructor() {
    super();
    attachTemplate(this);
    attachStyle(this);
  }
}
```

Set up the `Component` decorator, passing in the tag name as `in-td` to the `selector` property, the appropriate `Object` for registering the customized built-in element with the `CustomElementRegistry`. Leave the `style` and `template` properties blank for now.

```
@Component({
  selector: 'in-td',
  custom: { extends: 'td' },
  style: css`

  `,
  template: html`

  `,
})
export class TdComponent extends HTMLTableCellElement {
```

`TableComponent` should display two modes: read-only and editable. In read-only mode, each `TdComponent` should display non-editable read-only text, while in edit mode the component should display a text input, more specifically `TextInputComponent`. Add two `HTMLSpanElement` to the template, tagging each with a unique CSS class name. In the span classed `td-input`, add an instance of `TextInputComponent`.

```
  template: html`
    <span class="td-readonly"></span>
    <span class="td-input">
      <in-texinput type="text"></in-texinput>
    </span>
  `,
```

Before proceeding with styling `TdComponent`, head back on over to `Table.stories.js` and import `TextInputComponent` into the file.

```
import { TextInputComponent } from "../input/TextInput";
```

Export `TextInputComponent` from `Table.stories.js` to satisfy Storybook's Webpack compilation.

```
export {
  CardComponent,
  TableCardComponent,
  TableComponent,
  TrComponent,
  TdComponent,
  TextInputComponent,
  ColumnData,
};
```

Moving back to `TdComponent`, add styling for both the `HTMLSpanElement` you introduced in the template. We need to ensure the read-only content is styled per the spec in Figma, while also retaining the characteristics of a `HTMLTableCellElement`. To hide/show the read-only value, let's use a similar method to the buttons in `TableCardComponent`. Add a style for when the `hidden` HTML attribute is truthy. The element should be `display: none`.

```
style: css`
  .td-readonly {
    display: table-cell;
    box-sizing: border-box;
    margin-top: var(--margin-xs);
    padding-left: var(--padding-sm);
    height: 44px;
    vertical-align: middle;
  }
  .td-readonly[hidden='true'] {
    display: none;
  }
`,
```

Add getters for each of the elements you introduced to the layout: both `HTMLSpanElement` and the `TextInputComponent`. You'll need to reference each of these elements in the component logic you'll develop in `TdComponent`.

```
  get $readOnly(): HTMLSpanElement {
    return this.querySelector('.td-readonly');
  }

  get $input(): HTMLInputElement {
    return this.querySelector('in-texinput');
  }

  get $inputContainer(): HTMLSpanElement {
    return this.querySelector('.td-input');
  }
```

Make another method on `TdComponent` for setting the value of the read-only `HTMLSpanElement`. Pass an argument named `value` through the method and set `value` on the `innerText` of the `HTMLSpanElement`.

```
setValue(value: string) {
  this.$readOnly.innerText = value;
}
```

You'll need a way for `TableComponent` to communicate the value of the cell to `TdComponent`. Since you're already looping over `HTMLTableCellElement` in the context of `TableComponent` it seems appropriate to set the value via a HTML attribute. Listen for changes to a `value` attribute in `TdComponent` with the `observedAttributes` static getter.

```
static get observedAttributes() {
  return ['value'];
}
```

Declare the `attributeChangedCallback` method. While using a `switch` statement, watch for the `case` of `value` and call `setValue` there.

```
attributeChangedCallback(name, prev, next) {
  switch (name) {
    case 'value':
      this.setValue(next);
      break;
  }
}
```

We'll require another method for toggling the mode of the `TdComponent` from read-only to editable. Declare a method named `setMode` and pass an argument named `readOnly`, type defined as `boolean`, to an `if` statement. Depending on the truthiness of `readOnly`, hide or show the appropriate `HTMLSpanElement`.

```
setMode(readOnly: boolean) {
  if (readOnly) {
    this.$readOnly.removeAttribute('hidden');
    this.$inputContainer.setAttribute('hidden', 'true');
  } else {
    this.$readOnly.setAttribute('hidden', 'true');
    this.$inputContainer.removeAttribute('hidden');
  }
}
```

Add `readonly` to the `Array` of `observedAttributes` and make a new `case` in the `attributeChangedCallback`, being sure to call `setMode` with the value of the `readonly` attribute. Since HTML attributes are `String`, we need to coerce the value of the attribute to a boolean because `setMode` expects a `boolean`. You can do this with the expression `next === 'true'`, which will return a `boolean` that is truthy, otherwise falsy.

```
static get observedAttributes() {
  return ['value', 'readonly'];
}

attributeChangedCallback(name, prev, next) {
  switch (name) {
    case 'value':
      this.setValue(next);
      break;
    case 'readonly':
      this.setMode(next === 'true');
      break;
  }
}
```

`TdComponent` is now set up to render different user interfaces based on the mode passed through the `readonly` HTML attribute.

However; we still need to refactor `TableComponent` to accommodate the new customized built-in elements.

Rendering Modes

In the last section, we dynamically created `HTMLTableRowElement` and `HTMLTableCellElement` with `createElement` in the `renderRows` method in `TableComponent`. Since then, we made these elements customized built-in elements and need to update the logic in `renderRows` to accommodate the change. `createElement` takes a second argument, an `Object` that specifies for the browser that these should be instances of a customized built-in element with a tag name set on the `is` property.

Update the calls to `createElement` in `renderRows` to create customized built-in elements with the tag names you declared earlier in this section.

```
const tr = document.createElement("tr", { is: "in-tr" });
```

```
const td = document.createElement("td", { is: "in-td" });
```

The initial state of each table cell should be read-only as opposed to edit mode, so ensure each `TdComponent` has a call to `setAttribute` that sets the `readonly` HTML attribute to `true` on initial render.

```
td.setAttribute("value", rowData[colData.property]);
td.setAttribute("readonly", "true");
```

Using CustomEvent to Transmit Row Data

Thus far we haven't done much with `TrComponent`, even though the purpose of this component is to store row state. `TrComponent`

is a direct descendant of `TableComponent` and in the `renderRows` method we already have a reference to each `TrComponent`. One of the requirements for using `CustomEvent` is being able to easily select the element. The main reason we've stuck with `BroadcastChannel` is because the deeply nested elements that required events were not immediately queriable from higher up in DOM. Even then, we needed to use attribute drilling so the nested `TableComponent` could listen to the same channel. Although this only required passing a `String` through HTML attributes and not a complex `Object` or `Array`. `CustomEvent` can be a viable means for component communication, especially when complex data has to be transmitted.

The `Object` that corresponds to each row in the loop is `rowData`. Use the `dispatchEvent` on `HTMLElement` to send a `new CustomEvent` named `data`, passing in `rowData` in the `detail` property.

```
tr.dispatchEvent(
  new CustomEvent("data", {
    detail: rowData,
  })
);
```

Let's react to this `CustomEvent` in `TrComponent` before providing a method that displays each mode in `TableComponent`. In `Tr.ts`, add a new `public` property on the `class` called `$rowData`. This property should be publicly accessible so `TableComponent` can access it.

```
public $rowData: any;
```

Import `Listen` from `@in/common` to set up a new event listener.

```
import { Component, Listen } from "@in/common";
```

In `TableComponent`, you dispatched a `CustomEvent` to the instance of `TrComponent`, so let's declare a new event listener with `Listen`.

Declare a new method named `setValue` that serves as the event listener callback method. Pass the event through the first argument `ev` and set the value of `this.$rowData` to the `detail`.

```
@Listen('data')
setValue(ev: CustomEvent) {
  this.$rowData = ev.detail;
}
```

We'll circle back to `TrComponent` later, but for now, let's stick with managing the rendering of nodes in `TableComponent`. Make a new method on `TableComponent` named `onReadOnly`. the purpose of this method is to display the `TdComponent` in read-only mode. When the method is called, select all the cells with `querySelectorAll` and loop over the `NodeList` with `forEach`. On every iteration of the loop, set the `readonly` HTML attribute of each `TdComponent` to `true`.

```
onReadOnly() {
  const cells = this.querySelectorAll('td');
  cells.forEach((td) => {
    td.setAttribute('readonly', 'true');
  });
}
```

Table Modes and State

Read-only mode was fairly trivial. To set the display of each `TdComponent`, all you had to do was set the `readonly` HTML attribute. Edit mode is a bit more complex. After the user initiates edit mode by pressing the edit button, they have the option of editing each cell or canceling the edits they made. We need a way to store the state of the rows in the table when the user enters edit mode, in the event the user wants to revert the changes. This is why we went

to the trouble of storing the data for each row on `TrComponent`, but we still need a way to store the state of all rows in the context of `TableComponent`. To get started, import `TrComponent` into `Table.ts` because you'll need to reference the `$rowData` property that exists on each instance of `TrComponent` in upcoming logic.

```
import { TrComponent } from "./Tr";
```

Declare a getter method on `TableComponent` that queries for all `TrComponent` in the `TableBodyElement` using `querySelectorAll`. We don't need an `Array` of `TrComponent`, but instead an `Array` of row data, so we need to use a method on `Array.prototype` not available to us on the `NodeList`. To access `map` on a `NodeList` you have to coerse the `NodeList` into an `Array` using `Array.prototype.from`.

```
get state() {
  return Array.from(this.querySelector('tbody').querySele\
ctorAll('tr')).map(
    (tr: TrComponent) => tr.$rowData
  );
}
```

Add a new `private` method on `TableComponent` named `savedState`. We'll use this property to store the state of the rows when the user enters edit mode.

```
private savedState: any[];
```

Declare another method on `TableComponent` named `onEdit`. This method will handle the display of elements in edit mode and changes to row state. In `onEdit` check if `this.savedState` exists and if truthy, set `this.savedState` to the `Array` returned from the `this.state` getter. Wrap the `Array` in a call to `JSON.parse(JSON.stringify())` to copy the `Array` and make it ready to dispatch the saved state in a `CustomEvent` or message on `BroadcastChannel` if necessary.

```
onEdit() {
  if (!this.savedState) {
    this.savedState = JSON.parse(JSON.stringify(this.stat\
e));
  }
}
```

Update the `onReadOnly` method to check for `this.savedState` and if truthy, call `renderRows` with the data model stores in `savedState`, then reset `savedState`, making it equal to `undefined` for the event. If the user enters edit mode once again, the check for `savedState` in `onEdit` can execute as expected.

```
 onReadOnly() {
  const cells = this.querySelectorAll('td');
  cells.forEach((td) => {
    td.setAttribute('readonly', 'true');
  });
  if (this.savedState) {
    this.renderRows(this.savedState);
    this.savedState = undefined;
  }
}
```

Orchestrating Edit Mode

The acceptance criteria for the editable table suggests that users should be able to edit each cell in the table and either save or cancel the edit. We've chosen to create a separation of concerns where needed, placing row state on instances of `TrComponent`, the display state for each cell on `TdComponent`, and the overall rendering of rows and cells on `TableComponent`. This separation of concerns begins to break down when considering the complexity of edit mode. Some concerns regarding edit mode are below.

- When a user edits a field and moves on to another input, row state should be updated
- Upon using the keyboard to navigate, users should expect to use the `Tab` key to focus the next logical cell, i.e. when focused on the last cell of a row and the user presses `Tab`, focus should flow to the first cell of the next row or the buttons in the footer.

These user experience enhancements require some binding of event listeners on the `TextInputComponent` embedded in `TdComponent`. Some may argue that this is the concern of `TdComponent` so any event listener logic should be declared in the context of `TdComponent`. Another argument for handling this logic in `TableComponent` is that `TableComponent` is responsible for rendering of nodes and should track which nodes are currently being edited as well as rendered. For the sake of development, we'll take this stance, relegating `TdComponent` to merely being concerned with view logic, while `TableComponent` is responsible for tracking not only which rows and cells are rendered, but are also being edited. For this purpose, let's add a new property on `TableComponent` named `editIndex` and type define that property as `number`, then set the initial value to `0`; the index of the first cell in the table.

```
1  private editIndex: number = 0;
```

`TableComponent` can only interpret a cell is being edited when the user focuses on the `TextInputComponent` visible in edit mode. To listen for the focus event, we'll need to set up event listeners and react to events on each `TextInputComponent`. Import `TdComponent` into `Table.ts` to get started.

```
1  import { TdComponent } from "./Td";
```

Declare a new method on `TableComponent` named `handleCellListeners`. The first argument of this method should be

the `TdComponent` so we'll name it `td`, followed by the `index`, which has a type of `number`. Setup the method some more by making a new `const` named `tr`, selecting the `TrComponent` that is the parent of the `TdComponent` using the `parentNode` method on `HTMLElement`. Declare another `const` named `input` and use `querySelector` to select the `HTMLInputElement`. We could type define this input as `TextInputComponent` as well, but that isn't necessary since we are only going to reference methods found on `HTMLInputElement`.

```
handleCellListeners(td: TdComponent, index: number) {
  const tr = td.parentNode as TrComponent;
  const input = td.querySelector('in-texinput') as HTMLIn\
putElement;
}
```

In `onEdit`, select all the `HTMLTableCellElement` using `querySelectorAll` and using `forEach`, call `this.handleCellListeners` on the `Array` of `HTMLTableCellElement`, ensuring to bind the context to `this` (`TableComponent`).

```
onEdit() {
  const cells = this.querySelectorAll('td');
  if (!this.savedState) {
    this.savedState = JSON.parse(JSON.stringify(this.stat\
e));
  }
  cells.forEach(this.handleCellListeners.bind(this));
}
```

Back in `handleCellListeners`, check if the `input` exists with an `if` statement. This is the safest approach because later we may want to add elements other than `HTMLInputElement` in each `HTMLTableCellElement`. Declare three event listeners on the input, one for `onclick`, another for `onkeyup`, and the third for `onkeydown`.

```
if (input) {
  input.value = td.getAttribute("value");
  td.setAttribute("readonly", "false");

  input.onclick = (ev) => {};

  input.onkeyup = (ev) => {};

  input.onkeydown = (ev) => {};
}
```

In the `onclick` event listener, set `this.editIndex` to the index of the `HTMLTableCellElement` in the `Array` of all `HTMLTableCellElement` using `indexOf` on `Array.prototype`. You'll have to coerce the `NodeList` into an `Array` with `Array.prototype.from` because `NodeList` lacks many methods found on the `Array` prototype.

```
input.onclick = (ev) => {
  const cells = this.querySelectorAll("td");
  this.editIndex = Array.from(cells).indexOf(td);
};
```

Whenever the user edits the value of the `HTMLInputElement`, we want to update the value of the cell in state, which is currently stored in two places: on `TrComponent` and the `value` HTML attribute on `TdComponent`. We can listen for changes to the `HTMLInputElement` using the `onkeyup` event listener and set the value in both of these places, although we'll need to create a larger interface in both places. `TrComponent` has no way of listening for changes in row state and reacting to those changes.

The state of each row is an `Object` that contains multiple properties. Each property is mapped to a cell. In the context we are working in `handleCellListeners`, we have access to the `TrComponent` and `TdComponent`, although the `TdComponent` only stores the value of the

cell, not the property on the row state (`$rowData`) the cell is mapped to. To add the name of the property on `TdComponent`, we need to turn to the `renderRows` method, where the elements are rendered. Set a new HTML data attribute with the value of the property in the loop.

```
td.setAttribute("data-property", colData.property);
```

Returning to the `handleCellListeners`, declare a new event listener for `onkeyup`. Set the `value` HTML attribute on `TdComponent` using the current `input.value`. Dispatch a `CustomEvent` to the `TrComponent` with a new event name: `patch`, meaning we're going to update a property on the `Object` held in row state on `TrComponent`. In the `CustomEvent detail`, pass the value of the `TdComponent` as it was just set via the HTML attribute (or use the `input.value`) and the property set on the HTML data attribute `data-property`.

```
input.onkeyup = (ev) => {
  td.setAttribute("value", input.value);
  tr.dispatchEvent(
    new CustomEvent("patch", {
      detail: {
        property: td.getAttribute("data-property"),
        changes: td.getAttribute("value"),
      },
    })
  );
};
```

React to the new `CustomEvent` in `TrComponent` by adding a new method named `patchData` that includes the `Listen` decorator. Set the value of the property on `this.$rowData` to the appropriate value using the properties on the `CustomEvent detail`.

```
@Listen('patch')
patchData(ev: CustomEvent) {
  this.$rowData[ev.detail.property] = ev.detail.changes;
}
```

Back in the `handleCellListeners` method on `TableComponent` declare the last event listener, this time, listening for `onkeydown` events on the `HTMLInputElement`. We want to evoke `ev.stopPropagation()` here to prevent the browser default and insert a check if the user pressed `Tab`. When the user presses the Tab key we want to set the current `editIndex` to the index of the cell, which is available via `index`.

```
input.onkeydown = (ev) => {
  ev.stopPropagation();
  if (ev.key === "Tab") {
    this.editIndex = index;
  }
};
```

There's more to do here. When the user presses the Tab key, the focus should advance to the next editable input in the table. We need another method that handles this functionality because it will need to be evoked in multiple places to focus not just the next input, but also the first input when the user enters edit mode. For lack of a better name we'll call this method `onNext`.

In the `onNext` method, query for all the `TdComponent`. Check if a cell exists at the current index using the expression `!cells[this.editIndex]` and if this expression returns truthy, return so any further code is not executed on an element that does not exist. Next, select the `TdComponent` at the current index and query for the `TextInputComponent`. Checking again if the input exists, call `focus()` on the `HTMLInputElement`. This has the effect of focusing the next input in the table.

```
onNext() {
  const cells = this.querySelectorAll('td');
  if (!cells[this.editIndex]) {
    return;
  }
  const input = cells[this.editIndex].querySelector(
    'in-textinput'
  ) as HTMLInputElement;
  if (input) {
    input.focus();
  }
}
```

To focus the first cell when the user initiates edit mode, add a call to `onNext` in the `onEdit` method.

```
onEdit() {
  const cells = this.querySelectorAll('td');
  if (!this.savedState) {
    this.savedState = JSON.parse(JSON.stringify(this.stat\
e));
  }
  cells.forEach(this.handleCellListeners.bind(this));
  this.onNext();
}
```

Call `onNext` in the context of the `handleCellListeners` method after checking if the user pressed the Tab key.

```
input.onkeydown = (ev) => {
  ev.stopPropagation();
  if (ev.key === "Tab") {
    this.editIndex = index;
    this.onNext();
  }
};
```

To reset the `editIndex` back to the initial value of `0` before the user enters edit mode, you can hijack `onReadOnly` which occurs before `onEdit`. Set `editIndex` to `0` at the end of this method.

```
onReadOnly() {
  const cells = this.querySelectorAll('td');
  cells.forEach((td) => {
    td.setAttribute('readonly', 'true');
  });
  if (this.savedState) {
    this.renderRows(this.savedState);
    this.savedState = undefined;
  }
  this.editIndex = 0;
}
```

That's it for the functionality surrounding editing table cells in edit mode. So far you've tackled the the R (Read) and U (Update) in CRUD, but soon you'll work on the C (Create) and D (Delete). Before you get there, you'll add the functionality for the user to save edited fields and fix styling issues that have appeared in the `TableComponent`.

Save

We need to take care of some issues with saving data. After the user presses the save button, the table should revert to read-only

mode with all the changes the user made. We'll need to clean up the current view by setting the cell the user last edited to read-only mode and reinitialize `savedState` and `editIndex`. For convenience we can also post a message whenever the user saves on the `BroadcastChannel` which can be listened for outside of the context of `TableComponent`, like in the context of the Story or application source code. Lastly, we need to re-render the table in read-only mode.

```
onSave() {
  const data: TrComponent[] = this.state;

  if (this.querySelectorAll('td')[this.editIndex]) {
    this.querySelectorAll('td')[this.editIndex].setAttrib\
ute(
      'readonly',
      'true'
    );
  }

  this.savedState = undefined;
  this.editIndex = 0;

  this.channel.postMessage({
    type: 'change',
    detail: data,
  });

  this.renderRows(data);
}
```

Make a new method on `TableComponent` named `onSave`. Make a `const` named `data` equal to `this.state` and give it the proper type definition. Call `setAttribute` on the current `TdComponent` if it exists. Reset the values of `savedState` and `editIndex` to their defaults,

in case the user enters edit mode again. Post a message on the `BroadcastChannel`, this time setting the `type` property to `change` and pass the `data` to the `detail` property, `data` in this case being all the table rows serialized to JSON in the `state` getter. Finally, call `renderRows`, passing the method the `data` to render the table in read-only mode with the current state.

Fix Styling Issues

When developing the initial boilerplate for `TdComponent` we made a call to `attachStyle` in the `constructor`. Due to the way `TdComponent` are dynamically being added to DOM, a much safer implementation would be to call this `function` in the `connectedCallback`. `attachStyle` then appends a `<style>` tag to the nearest `ShadowRoot` or `document.head` to style the customized built-in element.

```
export class TdComponent extends HTMLTableCellElement {
  constructor() {
    super();
    attachTemplate(this);
  }

  connectedCallback() {
    attachStyle(this);
  }
```

After making `TableComponent` switch between two modes, some styling issues have cropped up. The `TableHeadElement` is misaligned with the rest of the layout. We just need to update the padding a tad to fix these issues.

In the `style` property on the `Component` decorator, add the following padding to the `TableHeadElement`:

```
padding-left: calc(var(--padding-xxs) + var(--padding-sm)\
);
padding-right: var(--padding-xxs);
```

When you are finished, the style declaration for the `TableHeadElement` should be as follows.

```
[is="in-table"] th {
  display: table-cell;
  box-sizing: border-box;
  margin-top: var(--margin-xs);
  padding-left: calc(var(--padding-xxs) + var(--padding-s\
m));
  padding-right: var(--padding-xxs);
  height: 44px;
  vertical-align: middle;
  font-weight: var(--font-weight-default);
}
```

Similarly, to match the Figma mockups, the padding of all cells in the table need to be normalized, with special styling for the first and last cells in each row.

```
[is="in-table"] td {
  padding-left: var(--padding-xxs);
  padding-right: var(--padding-xxs);
}
[is="in-table"] td:first-child {
  padding-left: var(--padding-lg);
}
[is="in-table"] th:first-child {
  padding-left: calc(var(--padding-lg) + var(--padding-sm\
));
}
[is="in-table"] th:last-child,
```

```
[is="in-table"] td:last-child {
  padding-right: var(--padding-lg);
}
```

In `TableCardComponent`, some other styling issues should be addressed. The button widths could be normalized with `min-width` and have even spacing between them using `margin-left`. The items in the footer need some adjustment. By giving the footer a `display:flex` and `justify-content: space-between`, we can set up the footer to accept a button on the left side as well as right. Additionally, we can give child elements that have the HTML attribute `hidden` the style of `display: none` to properly hide them.

```
:host .primary[is="in-button"],
:host .secondary[is="in-button"] {
  min-width: 160px;
  margin-left: var(--margin-lg);
}

.table-footer {
  display: flex;
  justify-content: space-between;
  padding-top: var(--padding-md);
}

[hidden] {
  display: none;
}
```

Now that styling issues are out of the way, it's time to focus on the C in CRUD: Create.

Create a Row

With the bulk of the logic finished for `TableComponent` we can now focus on the functionality for creating and deleting rows. According to the mockup, the user initiates the creation of a row by pressing a button in the `TableCardComponent` footer that looks like a "+". For the sake of the flexbox layout we just added to the footer, let's add a new container in the `TableCardComponent` template to encapsulate the button. Make a new `HTMLDivElement` and give it a CSS class name `crud-actions`.

```
<div class="crud-actions"></div>
```

Inside of this `HTMLDivElement` place the template of the `HTMLButtonElement` with an embedded SVG that displays a plus sign. If this seems cumbersome, that's perfectly fine. In the next chapter we'll streamline this process of adding complex complex layouts to a template using HTML templates. Until then, adding a static layout is a good solution.

```
<button class="icon icon-add button-add" is="in-button">
  <svg
    aria-hidden="true"
    focusable="false"
    data-prefix="fas"
    data-icon="plus"
    class="svg-inline--fa fa-plus fa-w-14"
    role="img"
    xmlns="http://www.w3.org/2000/svg"
    viewBox="0 0 448 512"
  >
    <path
      fill="currentColor"
      d="M416 208H272V64c0-17.67-14.33-32-32-32h-32c-17.6\
7 0-32 14.33-32 32v144H32c-17.67 0-32 14.33-32 32v32c0 17\
```

```
.67 14.33 32 32 32h144v144c0 17.67 14.33 32 32 32h32c17.6\
7 0 32-14.33 32-32V304h144c17.67 0 32-14.33 32-32v-32c0-1\
7.67-14.33-32-32-32z"
    ></path>
  </svg>
</button>
```

Add another getter on `TableCardComponent` to reference the button you just added to the template.

```
get $addButton(): HTMLElement {
  return this.shadowRoot.querySelector('.button-add');
}
```

Declare another method on `TableCardComponent` named `add`. This method will contain all the logic for adding a row in the context of `TableCardComponent`. Use the `Listen` decorator to set up an event listener for the `click` event on the element with the CSS class name `.button-add`. When the user clicks the add button, blur the button to allow focus to eventually travel to the first `HTMLInputElement` in the blank row that will be created. Hide and show the appropriate buttons in the `TableCardComponent` layout. Finally, post a message via `BroadcastChannel` with the type property set to `add`.

```
@Listen('click', '.button-add')
add() {
  this.$addButton.blur();
  this.$editButton.setAttribute('hidden', 'true');
  this.$saveButton.removeAttribute('hidden');
  this.$cancelButton.removeAttribute('hidden');
  this.channel.postMessage({
    type: 'add',
  });
}
```

In `TableComponent` react to the new message by adding a `case` in the `onMessage` method for `add` and call a new method yet to be declared: `onAdd`.

```
onMessage(ev) {
  switch (ev.data.type) {
    case 'add':
      this.onAdd();
      break;
```

Declare the new `onAdd` method on `TableComponent`.

```
onAdd() {

}
```

Similar to edit mode, we want to check if `savesState` exists and if not, set `savedState`.

```
if (!this.savedState) {
  this.savedState = JSON.parse(JSON.stringify(this.state)\
);
}
```

Currently state for rows is stored in the context of each `TrComponent`, but we haven't generated a new instance of `TrComponent` for the blank row of data yet. We can use a property on `TableComponent` to store a temporary `Object` that initializes the data for a blank row.

```
private blankRowData: any;
```

In `onAdd` set the `blankRowData` property to an empty `Object`. Loop over `columnData` and set the property associated with each column to a property name on `blankRowData`, making sure to set each property on `blankRowData` to an empty `string`.

```
this.blankRowData = {};
this.columnData.forEach((colData) => {
  this.blankRowData[colData.property] = "";
});
```

Make a new `const` named `rowData` and set it to `this.blankRowData`.

```
const rowData = this.blankRowData;
```

The rest of the algorithm looks a lot like `renderRows`. Generate a new `TrComponent`. Loop over the columns in `columnData` and on each iteration, generate a new `TdComponent`, being sure to initialize the appropriate HTML attributes on `TdComponent`. This time, set the `readonly` HTML attribute to `false` so the cells display with `TextInputComponent`. Finally, call `appendChild` on the `TableBodyElement` and pass in the `TrComponent` to display a blank row.

```
const tr = document.createElement("tr", { is: "in-tr" });
this.columnData.forEach((colData) => {
  const td = document.createElement("td", { is: "in-td" }\
);
  if (colData.align) {
    td.align = colData.align;
  }
  td.setAttribute("data-property", colData.property);
  td.setAttribute("readonly", "false");
  td.setAttribute("value", rowData[colData.property]);
  tr.appendChild(td);
});
this.$body.appendChild(tr);
```

Dispatch a new `CustomEvent` with type `data` to the `TrComponent`, being sure to set the `detail` with the `rowData`. This initializes the

`TrComponent` that was just appended to DOM with the correct state. Loop through the cells of the row and attach the event listeners to each cell with the `handleCellListeners` method. Set the edit index to the index of the first `TdComponent` in the blank row. Call `onNext` to focus the input in the row.

```
tr.dispatchEvent(
  new CustomEvent("data", {
    detail: rowData,
  })
);

const cells = this.querySelectorAll("td");

cells.forEach(this.handleCellListeners.bind(this));

this.editIndex = Array.from(cells).indexOf(
  tr.children[0] as HTMLTableCellElement
);

this.onNext();
```

You should now be able to observe in Storybook that when a user presses the add button, a blank row is created. Upon editing the row, the blank row behaves similar to existing rows thanks to the various methods on `TableComponent`.

Delete a Row

Looking at the mockup in Figma, it appears every row should have a delete button in the last cell of the row, on the right hand side. So far every cell has the same functionality: display read-only text or editable input. This cell is different. A button should be displayed. When the user presses the button, the row should be deleted. The delete button is only visible in edit mode.

The delete button will be visible in edit mode and should be visible when the user creates a new row. Let's abstract the generation of the DOM elements that make up the delete button cell in a method on `TableComponent` named `createDeleteButton`. This method will have to be called in `renderRows` and `onAdd`. In both contexts, we have access to the `TrComponent`.

In the new `createDeleteButton` pass in the `TrComponent` in the first argument of the method. The delete button needs to be encapsulated by a `HTMLTableCellElement`. Use `createElement` to generate a new instance of `HTMLTableCellElement`. Add a CSS class name with `classList.add` that differentiates this cell from other editable cells: `delete-cell`. Use `setAttribute` to set the `readonly` property of the cell to `true`. Set the `innerHTML` of the `HTMLTableCellElement` to a `ButtonComponent` with an inline SVG that displays the "X". Append the `HTMLTableCellElement` to the `TrComponent`.

```
 createDeleteButton(tr: TrComponent) {
    const deleteButtonTd = document.createElement('td');
    deleteButtonTd.classList.add('delete-cell');
    deleteButtonTd.setAttribute('readonly', 'true');
    deleteButtonId.innerHTML = `
    <button class="icon icon-trash" is="in-button">
      <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 \
0 448 512">
      <path
        fill="currentColor"
        d="M268 416h24a12 12 0 0 0 12-12V188a12 12 0 0 0-\
12-12h-24a12 12 0 0 0-12 12v216a12 12 0 0 0 12 12zM432 80\
h-82.41l-34-56.7A48 48 0 0 0 274.41 0H173.59a48 48 0 0 0-\
41.16 23.3L98.41 80H16A16 16 0 0 0 0 96v16a16 16 0 0 0 16\
 16h16v336a48 48 0 0 0 48 48h288a48 48 0 0 0 48-48V128h16\
a16 16 0 0 0 16-16V96a16 16 0 0 0-16-16zM171.84 50.91A6 6\
 0 0 1 177 48h94a6 6 0 0 1 5.15 2.91L293.61 80H154.39zM36\
8 464H80V128h288zm-212-48h24a12 12 0 0 0 12-12V188a12 12 \
0 0 0-12-12h-24a12 12 0 0 0-12 12v216a12 12 0 0 0 12 12z"
```

```
      />
    </svg>
  </button>`;
    tr.appendChild(deleteButtonTd);
  }
```

The delete button should be rendered in whenever rows are generated after all other `TdComponent` have been appended to the `TrComponent` The delete button should be the last cell displayed in each row. Just before the `TrComponent` is appended to the `TableBodyElement`, call `createDeleteButton`, passing the instance of `TrComponent` into the method's arguments.

```
this.createDeleteButton(tr);
```

Repeat this same step in the `onAdd` method.

In Storybook you should now observe a delete button at the end of each row in edit mode or when you add a new row, although the buttons are also visible in read-only mode. Let's fix this with some additional styling. In `TableComponent`, add a style that targets the element with CSS class name `.delete-cell` and `readonly` HTML attribute set to `true` and style the element with `display: none` so it's hidden in read-only mode.

```
[is="in-table"] td.delete-cell[readonly="true"] {
  display: none;
}
```

Fix the position of the button with a slight adjustment to the vertical position of the element using a `transform`.

```
[is="in-table"] td.delete-cell button {
  transform: translateY(-2px);
}
```

The buttons are not visible, but we haven't done anything to react to when the user clicks the button. The `handleCellListeners` method on `TableComponent` seems like an appropriate place. Near the start of the method, select the button by querying for an instance of `ButtonComponent`.

```
const button = td.querySelector('[is="in-button"]') as HT\
MLButtonElement;
```

Just like we checked if an `input` existed, establish an `if` statement that checks for the `ButtonComponent`. If the `ButtonComponent` exists it means this `TdComponent` has the `readonly` HTML attribute set to `true`, so toggle that to `true` for edit mode.

Set up an event listener for `onclick` and in the event listener callback, set the `editIndex` on the current cell in case the user wants to keep editing other fields. Dispatch a `CustomEvent` on the `TrComponent` with the event name `delete`. We'll set this up in a moment in `TrComponent`, but the intent is to send an event to the row to destroy itself. Add a second event listener for `onkeydown` so users triggering the button with a keyboard can have the desired effect. Call `ev.stopPropagation()` to prevent the browser default. Check if the key pressed was Tab and if so, navigate the focus to the next row by setting the `editIndex` to the current index and calling `onNext`. When the user presses Enter or Select, do the same thing, except this time dispatch the same `CustomEvent` to the `TrComponent`.

```
if (button) {
  td.setAttribute("readonly", "false");
  button.onclick = () => {
    this.editIndex = index;
    td.parentNode.dispatchEvent(new CustomEvent("delete")\
);
  };
  button.onkeydown = (ev) => {
    ev.stopPropagation();
    if (ev.key === "Tab") {
      this.editIndex = index;
      this.onNext();
    }
    if (ev.key === "Enter" || ev.key === "Select") {
      this.editIndex = index;
      this.onNext();
      td.parentNode.dispatchEvent(new CustomEvent("delete\
"));
    }
  };
}
```

In `TrComponent`, set up an event listener using the `Listen` decorator and a new method named `delete`. Inside this method call `removeChild` on the `parentNode` of the instance of `TrComponent` using the current context (`this`).

```
@Listen('delete')
delete() {
  this.parentNode.removeChild(this);
}
```

In Storybook you should now observe that when the user clicks on the delete button of each row, the row is deleted. If the user presses save, the current state of the table is retained.

Testing the Table

Integration tests are important for `TableComponent` mainly because there are several components that should work harmoniously together. With so many touch points, it's easy for compound components to break down. In this section, you'll test if the table can enter read-only mode, validate if the user can edit a text field, and also if the user can cancel edit mode. This should provide a comprehensive set of integration tests that could be further developed upon for more discrete test cases.

Integration Tests

For the integration tests, you'll be using the `Table` Story from the previous section as a test bed.

Make a new file in the `cypress/integration` directory named `Table.spec.js`.

```
touch cypress/integration/Table.spec.js
```

Add a `describe` block to the test file and use a `beforeEach` to mitigate any issues with testing the components in Storybook.

```
describe("Table Component", () => {
  beforeEach(() => {
    cy.wait(100);
  });
});
```

First let's test if edit mode can be initiated in the table. After the `beforeEach` make your first assertion and tell cypress to visit the appropriate route in Storybook.

Should Enter Edit Mode

```
it("should enter edit mode", () => {
  cy.visit("iframe.html?id=components-table--primary");
});
```

Using Cypress, target the root element in Storybook's iframe, then drill down to find the element with HTML attribute `[channel="table:one"]`, which should be the instance of `TableCardComponent`. Use the `shadow` method to target the autonomous custom element's Shadow DOM and the `find` method to select the footer `slot`. Chain the `find` method again to select the edit button and call `click()`. This should simulate when the user presses the edit button in the test environment.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="footer"]')
      .find(".button-edit")
      .click();
```

To validate the table is in edit mode after clicking the edit button, we're going to check that `TextInputComponent` exists in the first cell, which would otherwise be hidden in read-only mode. Target the root element again and drill down into the Shadow Root of the `TableCardComponent`. This time, select the content slot with `[slot="content"]` and find the `TableBodyElement`. Since we're working with customized built-in elements here, querying is much simpler. Drill down using consecutive `find` calls until you select the `TextInputComponent` and check if the element exists.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .find("td")
      .find("in-input")
      .should("exist");
```

The next test validates that the user can edit a cell with `TextInputComponent`, click the save button, and the table retains the value of the field in read-only mode. Set up a new test and visit the same route.

Should Edit a Text Field

```
it("should edit a text field", () => {
  cy.visit("iframe.html?id=components-table--primary");
});
```

The start of the test is similar to the first. The test should click the edit button to initiate edit mode.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="footer"]')
      .find(".button-edit")
      .click();
```

This time instead of checking `TextInputComponent` exists, call `click` on the input to bring it into focus, query against the `TextInputComponent` Shadow DOM with `shadow()` and find the

internal `HTMLInputElement` and clear it. This is to prevent issues with testing in various environments.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("in-input")
      .click()
      .shadow()
      .find("input")
      .clear();
```

For the next part of the test, drill down exactly the same way, only now call `type` on the `HTMLInputElement` and specify a string Cypress should insert.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("in-input")
      .click()
      .shadow()
      .find("input")
      .type("jane@doe.com");
```

Click the `TextInputComponent` again to bring it back into focus. This is to prevent issues with Cypress trying to focus on the save button in the next step.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .eq(2)
      .find("in-input")
      .click();
```

Wait a second before clicking the save button. This should make the test more reliable.

```
    cy.wait(1000);
```

Select the footer slot of `TableCardComponent` again and select the save button this time. Click the save button with Cypress.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="footer"]')
      .find(".button-save")
      .click();
```

Wait another second to improve reliability of the test.

```
    cy.wait(1000);
```

The table should now be in read-only mode, so you'll need to query against the `TdComponent` as if they only display a `HTMLSpanElement`. Assert the `HTMLSpanElement` contains the same text that was previously entered in the `TextInputComponent`.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("span")
      .contains("jane@doe.com");
```

The next test validates the user can cancel editing and the table reverts to the previous value of a cell. Establish a third test using the `it` keyword and have Cypress visit the same story.

Should Cancel Edit Mode

```
it("should cancel edit mode", () => {
  cy.visit("iframe.html?id=components-table--primary");
});
```

Just like the previous tests, click the edit button with Cypress.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="footer"]')
      .find(".button-edit")
      .click();
```

Clear the `HTMLInputElement` to reduce issues with testing.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("in-input")
      .click()
      .shadow()
      .find("input")
      .clear();
```

Set the value of the `HTMLInputElement` to the string `'jane@doe.com'`.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("in-input")
      .click()
      .shadow()
      .find("input")
      .type("jane@doe.com");
```

Use Cypress to click the cancel button.

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="footer"]')
      .find(".button-cancel")
      .click();
```

Validate the same cell `contains` the default value prior to editing: `'joe@gmail.com'`

```
    cy.get("#root")
      .get('[channel="table:one"]')
      .shadow()
      .find('[slot="content"]')
      .find("tbody")
      .find("tr")
      .first()
      .find("td")
      .first()
      .find("span")
      .contains("joe@gmail.com");
```

That's it for basic integration tests for the editable table. We could validate more test cases like adding and deleting rows or edge cases but for now we'll leave the tests here to concentrate on other material.

Snapshots

With Storybook still running, generate snapshots for the component.

```
yarn test:snapshot:ci
```

When snapshots have been generated for the Table story, add them to your commit. Commit your changes in git to complete this section. After you've completed integration and snapshot tests for `TableComponent` let's review the content of this chapter.

Summary

The editable table is a compound component that requires several discrete units of functionality to work seamlessly together in order

to provide a rich user experience. In this chapter, you coded four different components: `TableCardComponent`, `TableComponent`, `TrComponent`, and `TdComponent` while using Storybook as both development environment and test bed. Compound components demand some architectural design, otherwise maintenance can become a nightmare. By abstracting reusable code into `class` and `method`, we were able to keep the implementation fairly DRY (Don't Repeat Yourself). A strict separation of concerns was further provided by the library developed in the previous chapter. Since high level component functionality was abstracted to library functions, we focused mainly on developing the features required by the editable table experience.

Compound components have several ways to communicate. HTML attributes, `CustomEvent` and `BroadcastChannel` all have their relative strengths and weaknesses. In this chapter, we used all three methods in the development of the table. Knowing whether to use one over the other can make or break the developer experience or cause performance issues for the end-user (forgetting to remove event listeners, for instance). HTML attributes and `CustomEvent` may have made a lot of sense but `BroadcastChannel` may have seemed out of place. `BroadcastChannel`[64] was originally intended to provide a messaging system for communication across browser contexts. Using `BroadcastChannel` to overcome the limitations of event bubbling is an interesting solution, however, another specification that allows custom elements to pass data between components may be on the horizon. The Context API Proposal[65] would replace the need for `BroadcastChannel` if the proposal makes it's way into mainstream browsers.

We coded autonomous custom elements and customized built-in elements where it made sense. An autonomous custom element was reserved for the `TableCardComponent`. `TableComponent`, `TrComponent` and `TdComponent` are customized built-in elements so they retain

[64] https://developer.mozilla.org/en-US/docs/Web/API/BroadcastChannel

[65] https://github.com/webcomponents-cg/community-protocols/blob/main/proposals/context.md

the characteristics of the table elements they extend from. We enhanced the typical HTML table by making the cells editable and providing the ability to add and delete rows, while styling the table per the mockup in Figma.

If there was one unrelated nuisance in `TableComponent`, it would be the inline SVGs that bloat the template strings. Suppose designers wanted to update the icons used in the add and delete buttons. It would mean tracking down template strings where these SVGs exist in the codebase. HTML templates could make the SVG implementation in `TableComponent` easier to maintain over time if they were abstracted away from the component implementation and stored elsewhere. In the next chapter you'll unlock the magic of HTML templates while developing the `DialogComponent`.

Powering Dialogs with HTML Templates

Chapter Six

HTML Templates

Programmatically making a template used to be difficult. Templating HTML was either the purview of ES2015 template strings or the domain of JavaScript frameworks. Either way, HTML templates were bundled into JavaScript, separated from the DOM where they were native. This is even the case in the UI library that you've been working on in the course of this book. HTML templates are declared in ES2015 template strings and packaged with the component.

```
const shadowRoot = this.attachShadow({ mode: "open" });
const template = document.createElement("template");
template.innerHTML = `
<div class="control">
  <input type="text" />
</div>
<div class="message"></div>
`;
shadowRoot.appendChild(template.content.cloneNode(true));
```

When creating a HTML template dynamically with JavaScript, you used `document.createElement` like you would when instantiating any DOM node, passing in the `string` `template` to hold a reference to a `HTMLTemplateElement`. You then set the `innerHTML`

of the `HTMLTemplateElement` to a custom template declared as an ES2015 template string. `HTMLTemplateElement` differ itself from other DOM nodes in at least two important ways. Instances of `HTMLTemplateElement` have a property named `content`, which is unlike other DOM nodes which lack this property. The `cloneNode` method on `content` allows you to copy the HTML template that was originally set with `innerHTML` and use it in the context of other methods like `appendChild`. You can essentially reuse HTML templates as if they were a rubber stamp. Each time `cloneNode` is called in the context of `appendChild` you are stamping the HTML template.

So far, we haven't harnessed the full potential of HTML templates. Instances of `HTMLTemplateElement` declared in DOM (not created dynamically like in the above example) are not parsed by the user agent, they are essentially hidden from users. You could place an HTML template in the document and the elements contained within the template won't be rendered until the template is cloned. Below is the same HTML template as it would appear if in the DOM.

```
<template>
  <div class="control">
    <input type="text" />
  </div>
  <div class="message"></div>
</template>
```

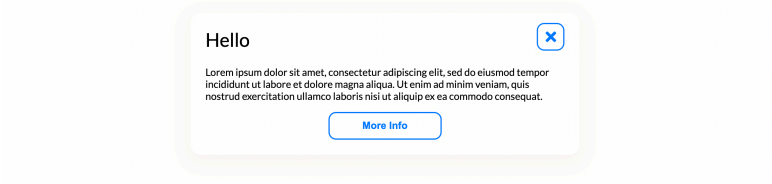
Using DOM to store HTML templates seems natural. When server-side rendered (SSR), HTML templates can be a powerful means to reuse templates. If SSR were implemented with some sort of metadata engine, each SSR rendered page could render initially with the HTML templates required for the user experiences on that page. In the context of single-page applications (SPA), the appeal of HTML templates may not seem obvious. Templates may be bundled with the SPA and parsed as JavaScript, although this bloats the

JavaScript bundle. What if HTML templates could be attached to each route via the SPA router and appended to DOM so they could be reused? Perhaps there are templates in a global scope of an application, like in the case of a "discard changes" or "are you sure?" dialog, where the user must confirm they want to proceed. The HTML template could be stored in DOM and reused throughout the application.

Dialogs seem like a good fit for HTML templates. If we coded a dialog custom element, it would be tempting to provide an inline template, but we could code the dialog in such a way that queries for HTML templates that exist in DOM and reuses them. This way, the dialog would only be concerned with positioning, or in the case of multiple instances of the dialog, the layer in which the dialog is drawn. By creating a separation of concerns between template and dialog, we can reduce code duplication in an application. The HTML template could be declared once with an identifier and instances of dialogs could query for the template and clone template content upon instantiation.

In this chapter, you'll build a `DialogComponent` that manages the state of a dialog instance (open or closed), the type of dialog that should be instantiated, and the dialog's relationship with the DOM element that triggers the dialog to open (an instance of `ButtonComponent` or `HTMLAnchorElement` most likely). This new component will query for HTML template and pass a clone of the template to the custom element that should be displayed. `DialogComponent` won't handle any display characteristics of the displayed user interface of a dialog, the instance of the `DialogComponent` will essentially be hidden from the user experience. In this way, this is the first time we're using a custom element `class` in the role of a "controller"; a `class` that updates both "model" and "view" in the traditional sense of MVC. `DialogComponent` doesn't have to be a custom element, although by making it one we can give engineers the ability to declare instances of `DialogComponent` in a template, which may seem more

intuitive.



There are two types of known dialogs in the application: modals and tooltips. Each type of dialog distinguishes itself by how the dialog is displayed. Since `DialogComponent` is not concerned with any display characteristics, we'll create two other autonomous custom elements that handle the display and positioning of a modal or tooltip. Modals are typically displayed centered above all other content, while tooltips are displayed contextually to the element the user clicks to instantiate an instance of the tooltip, just beneath the cursor. You can review the dialogs in Figma[66].

While developing each dialog, we'll come across an interesting use case where a dialog is embedded in the template of another dialog. When the user clicks on a link in a modal, a tooltip should appear, layered above the modal, just below the position of the `HTMLAnchorElement` the user clicked on. If the user closes the modal by clicking the close button or pressing the Escape key on a keyboard, both the tooltip and modal should close. This means we need some sort of controller that tracks the parent/child relationships of dialogs and the order the dialogs appear in the view. `DialogComponent` won't cut it for this task because the custom element isn't necessarily global or available during the lifespan of all dialogs that could be stacked on top of one another. `DialogComponent` is only concerned with the lifecycle of a single dialog. To handle the orchestration of all dialogs in a view, we'll code a `DialogStack class` that will exist on the `window`, or could be declared elsewhere in a global scope.

[66] https://www.figma.com/file/QXGa6qN6AqgeerCtS28I8z/Web-Components-Book-Design-Library?node-id=725%3A505

What you will build

- `DialogComponent` which handles the lifecycle of a single dialog
- `ModalComponent` which displays a model centered in the view
- `TooltipComponent` which displays a contextual tooltip just beneath an element
- `DialogStack` which orchestrates the state of all dialogs in a view

What you will learn

- How to unlock the potential of HTML templates
- How to use a custom element as a "controller"
- How to create and destroy custom elements on the fly
- How to create a custom dialog that displays anywhere
- How to control the order and position of dialogs
- How to reliably query DOM across shadow boundaries
- How to use data attributes to store information

Acceptance criteria

- `DialogComponent` handles state of a single dialog
- `DialogComponent` attaches an event listener to the DOM element that triggers opening a dialog on click
- `DialogComponent` calls methods on `DialogStack` to register dialogs and remove templates
- `ModalComponent` handles display and positioning of a modal
- `TooltipComponent` handles display and position of a tooltip
- `DialogStack` handles the order, focus, and storage of template

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-6 from where you left off at the end of chapter 5.

```
git checkout -b chapter-6
```

Mocking the Dialog in Storybook

Just like in previous chapters, we'll start by creating a directory that stores all the files associated with dialogs. In a new file named `Dialog.stories.js` we'll explore the outward facing API of `DialogComponent`: a set of HTML attributes. We'll scaffold `DialogComponent` in a separate file. By the end of this section you should have a development environment in Storybook that allows you to develop `DialogComponent`.

Define the Story

Make a new directory named `dialog` that's a sibling to other components.

```
mkdir packages/component/src/dialog
```

Create a new file named `Dialog.stories.js` in the dialog directory you just created.

```
touch packages/component/src/dialog/Dialog.stories.js
```

In `Dialog.stories.js` export the configuration in Component Story Format. Storybook needs to display a new story in the sidebar. The `title` of this Story is `Components/Dialog`. The tag name we'll use for `DialogComponent` will be `in-dialog`, which we can set on the `component` property. If you haven't already, start up Storybook with `yarn storybook`.

```
export default {
  title: "Components/Dialog",
  component: "in-dialog",
};
```

Import the `html function` from the lit-html package. We'll use this in the Story template.

```
import { html } from "lit-html";
```

Declare a new `const` called `DialogTemplate` and make it equal to a `function` that returns `html`. We can pass a template string into `html` first argument using shorthand syntax. In the template string, add a new tag named `in-dialog` and an instance of `ButtonComponent` that displays a primary button with the call to action "Open".

```
const DialogTemplate = () => {
  return html`
    <in-dialog></in-dialog>
    <button is="in-button" class="primary">Open</button>
  `;
};
```

Import `ButtonComponent` into the story file so it will appear in Storybook.

```
import { ButtonComponent } from "./../button/Button";
```

Export `ButtonComponent` from `Dialog.stories.js` to satisfy Webpack compilation in Storybook.

```
export { ButtonComponent };
```

Declare a new `const` named `Primary` and make it equal to a call to `DialogTemplate` using `bind`. Add a line so you can set `args` on the `Primary` story.

```
export const Primary = DialogTemplate.bind({});
Primary.args = {};
```

DialogComponent

We need to at the very least provide a boilerplate custom element of `DialogComponent` for the story. This will also enable development of `DialogComponent` in the next section. Make a new file named `Dialog.ts` in the same directory as the story.

```
touch packages/component/src/dialog/Dialog.ts
```

Import the necessary reusable `function` from `@in/common`. For the autonomous custom element you're about to develop, you will need `Component` and `attachShadow`.

```
import { Component, attachShadow } from "@in/common";
```

Export a `class` named `DialogComponent` that extends `HTMLElement`, being sure to call `super` and `attachShadow` in the `constructor`. This will give us a Shadow DOM template we can work with in the `Component` decorator.

```
export class DialogComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this, {
      mode: "open",
    });
  }
}
```

Decorate the `class` with the `Component` decorator, setting the `selector` to `in-dialog`. Give the component all the styling it requires via the `style` property. Since `DialogComponent` will have no visual component, we can safely set `display: none` on the `:host`.

```
@Component({
  selector: 'in-dialog',
  style: `
    :host {
      display: none;
    }
  `,
})
```

Import `DialogComponent` into `Dialog.stories.js` so we can use it in the context of the story.

```
import { DialogComponent } from "./Dialog";
```

Export `DialogComponent` from `Dialog.stories.js`.

```
export { ButtonComponent, DialogComponent };
```

That's all the `DialogComponent` boilerplate we need for now. In the next section, we'll focus on this `class` implementation.

Binding Mock Data with HTML Attributes

In `Dialog.stories.js` let's figure out the outward facing APIs we'll need so a user can click on the `ButtonComponent` to open a dialog. Most importantly, we need a stable API that can be applied to any element the user can click and the instance of `DialogComponent`. This could be achieved many ways, but we're going to choose a method that works via the template. HTML data attributes present an easy way to set custom attributes on an element, so let's opt to use them to store an identifier for the dialog on the `ButtonComponent`. If we have a predictable id specified on the `ButtonComponent`, we have a way to query for that element. Give the `args` a new property named `targetId` set to `dialog-hello-target`. From here forward, we'll refer to the DOM element that triggers the dialog to open as the "target".

```
Primary.args = {
  targetId: "dialog-hello-target",
};
```

Pass the `targetId` through `DialogTemplate` so you can set a new HTML `data-dialog-id` data attribute on the `ButtonComponent`.

```
const DialogTemplate = ({ targetId }) => {
  return html`
    <in-dialog></in-dialog>
    <button data-dialog-id="${targetId}" is="in-button" c\
lass="primary">
      Open
    </button>
  `;
};
```

We could strictly enforce that any target should be specified with the `data-dialog-id` HTML data attribute. However; it seems more

likely engineers may need more than one way to select the target in order to query for it in DOM. This query will happen in `DialogComponent`, where we'll attach and detach event listeners on the target. Add a new property on `args` named `targetSelector` and set it to the selector needed to query for the `ButtonComponent`.

```
Primary.args = {
  targetId: "dialog-hello-target",
  targetSelector: '[data-dialog-id="dialog-hello-target"]\
',
};
```

Just like before, pass the `targetSelector` through `DialogTemplate`, this time, setting a HTML attribute named `target` on the `DialogComponent`.

```
const DialogTemplate = ({ targetId, targetSelector }) => {
  return html`
    <in-dialog target="${targetSelector}"></in-dialog>
    <button data-dialog-id="${targetId}" is="in-button" c\
lass="primary">
      Open
    </button>
  `;
};
```

Each dialog is associated not just with a target, but also a HTML template. HTML templates are available in the Storybook iframe and just need to be selected via a query to be used. Each `HTMLTemplateElement` has a HTML data attribute declared on the template named `data-template-id`. The first template we'll reference is named `dialog-hello`. Add another property to `args` named `templateSelector` and set it to `[data-template-id="dialog-hello"]`.

```
Primary.args = {
  targetId: "dialog-hello-target",
  targetSelector: '[data-dialog-id="dialog-hello-target"]\
',
  templateSelector: '[data-template-id="dialog-hello"]',
};
```

Pass the `templateSelector` through the `DialogTemplate` first argument and set a new HTML attribute on the `DialogComponent` named `template`.

```
const DialogTemplate = ({ targetId, targetSelector, templ\
ateSelector }) => {
  return html`
    <in-dialog
      target="${targetSelector}"
      template="${templateSelector}"
    ></in-dialog>
    <button data-dialog-id="${targetId}" is="in-button" c\
lass="primary">
      Open
    </button>
  `;
};
```

Finally, we just need to tell `DialogComponent` which type of dialog to display: a modal or tooltip. The first example is a modal, so let's set another property on `args` named `variant` to `modal`.

```
Primary.args = {
  targetId: "dialog-hello-target",
  targetSelector: '[data-dialog-id="dialog-hello-target"]\
',
  templateSelector: '[data-template-id="dialog-hello"]',
  variant: "modal",
};
```

Pass `variant` to the `DialogComponent` by setting a new HTML attribute named `variant`.

```
const DialogTemplate = ({
  targetId,
  targetSelector,
  templateSelector,
  variant,
}) => {
  return html`
    <in-dialog
      target="${targetSelector}"
      template="${templateSelector}"
      variant="${variant}"
    ></in-dialog>
    <button data-dialog-id="${targetId}" is="in-button" c\
lass="primary">
      Open
    </button>
  `;
};
```

You now have a development environment suitable for developing the `DialogComponent`. In this section, we set up the story template with an instance of `DialogComponent` and `ButtonComponent` and gave each HTML attributes to link a target with a given dialog. If it still isn't clear what all these HTML attributes will be used for,

have no fear! In the next section, you'll develop `DialogComponent` and use the value of each of these attributes which should reveal their meaning. By declaring `DialogComponent` aside the target `ButtonComponent` in a template, we've hopefully given engineers an easy-to-use API to use throughout an application.

DialogComponent

This autonomous custom element is unlike others we've developed so far in that there is no view associated with the element. `DialogComponent` behaves more like a controller in the traditional Model View Controller (MVC) paradigm. This component will pass the model from the values of HTML attributes to the view (a modal), which it will also instantiate. The `DialogComponent` is already a `class` which is the ideal interface for a controller. We can store state for the individual dialog here with some `class` properties. We'll need to store whether the dialog is opened or closed. Set a new property named `$state` and type define this as `open | close`, a strict type definition which means the value of `$state` can only be one of two strings: `open` or `close`. Set the type definitions for properties associated with the three HTML attributes we set in the previous section: `$target`, `$targetSelector`, and `$templateSelector`. Lastly, set a property named `$target` we'll eventually use to store a reference of the `ButtonComponent` (or any other target).

```
1  public $state: 'open' | 'close';
2  public $target: Element;
3  public $targetSelector: string;
4  public $templateSelector: string;
5  public $variant: 'modal' | 'tooltip';
```

Observing Attributes

Declare the `observedAttributes static` getter on the `class` and listen for the three HTML attributes: `target`, `template`, and `variant`.

```
static get observedAttributes() {
  return ['target', 'template', 'variant'];
}
```

Set up the `attributeChangedCallback` with the usual arguments: `name`, `prev` and `next` and a `switch` statement. If the `case` the `name` matches `template`, set the value of `$templateSelector`. In the case of `variant`, set the value of `$variant`. You'll reference these properties later in the component logic.

```
attributeChangedCallback(name, prev, next) {
    switch (name) {
        case 'template':
        this.$templateSelector = next;
        break;
        case 'variant':
        this.$variant = next;
        break;
    }
}
```

To associate the dialog with a target, declare a new method named `setTarget`, which requires a single argument named `selector` typed as a `string`. This will be the selector needed to query for the `ButtonComponent`.

```
setTarget(selector: string) {

}
```

Update the `switch` in the `attributeChangedCallback` to call `setTarget` when the `case` matches `target`. This may have to be wrapped in a `setTimeout` to work properly in the context of the LitHTML `html` function. We need to wait for a tick of the JavaScript engine before querying for the element, which is achievable by passing `0` to the second argument of `setTarget`.

```
attributeChangedCallback(name, prev, next) {
    switch (name) {
        case 'target':
        setTimeout(() => {
          this.setTarget(next);
        }, 0);
        break;
        case 'template':
        this.$templateSelector = next;
        break;
        case 'variant':
        this.$variant = next;
        break;
    }
}
```

Setting the Target

In the `setTarget` method, set the value of the `$targetSelector` property you declared earlier on the `class` type definition. Query for the `ButtonComponent` using `document.querySelector` and set the reference of the `ButtonComponent` to `$target` for use later in this method.

```
this.$targetSelector = selector;
this.$target = document.querySelector(this.$targetSelecto\
r);
```

First thing we'll do in this method is add some error handling for when the target is not found via `querySelector`. If this is the case, we want the method to fail silently for the end-user, so we'll `return`, but only after notifying a developer of the problem via `console.error`.

```
if (!this.$target) {
  console.error(
    `DialogComponent cannot find Element with selector ${\
selector}`
  );
  return;
}
```

The entire reason we're querying for the `ButtonComponent` is so we can attach an event listener on click that will change `$state` from `close` to `open`. Declare a new method on `DialogComponent` for this purpose named `targetListener`. We may need access to the event, so add the first argument type defined as `MouseEvent`. If `$state` doesn't equal `open`, call another method that doesn't exist yet: `onOpen`.

```
targetListener(ev: MouseEvent) {
    if (this.$state !== 'open') {
        this.onOpen();
    }
}
```

Declare `onOpen` on `DialogComponent` and add a `console.log` here temporarily for testing out the method works.

```
onOpen() {
    console.log('open!');
}
```

In the `setTarget` method, call `addEventListener` on `$target`, being sure to listen for the `click` event. Bind the event listener callback to `targetListener` using `bind`. Optionally, set the capture to `false` in the third argument of `addEventListener`, which should be the default anyways.

```
this.$target.addEventListener("click", this.targetListene\
r.bind(this), false);
```

You should be able to observe in Storybook that a console message is logged when clicking the `ButtonComponent`. Remove that `console.log` in `onOpen` when you've verified that the logic is working.

Displaying the HTML Template

The purpose of `DialogComponent` is to take a model provided by HTML attributes and display a dialog. The content of the dialog is provided by a HTML template already in DOM. Previously you provided a selector for that template via HTML attribute to set the value of the `$templateSelector` property. We're going to return to the `onOpen` method to query for a HTML template with this selector in a moment, but first let's declare a new property named `$container` on the `class` where we can store a reference to the instance of the dialog that will be displayed.

```
public $container: Element;
```

In the `onOpen` method, set a local `const` named `template` to a reference of the `HTMLTemplateElement`. Use `$templateSelector` when querying for the template on the `document`.

```
const template = document.querySelector(
  this.$templateSelector
) as HTMLTemplateElement;
```

Similar to the `setTarget` method, let's add some error handling here for the use case that the HTML template does not exist. Provide some feedback for an engineer with `console.error` and use `return` if `template` is `undefined` so the rest of the method's logic can't run and cause a bad experience for the end-user.

```
if (!template) {
  console.error(
    `DialogComponent cannot find HTMLTemplateElement with\
 selector ${this.$templateSelector}`
  );
  return;
}
```

Eventually `$container` will be set to instances of `ModalComponent` or `TooltipComponent`, but we have yet to develop those components. For now use `createElement` to instantiate a new `HTMLDivElement` and set `$container` with that. Clone the template using `cloneNode` and set a new `const` named `clone` to the cloned template. So we verify the variant can be correctly passed, set a new CSS classname equal to `$variant` on the `$container`. Set the content of the `$container`, passing the `clone` to `appendChild`. Call `appendChild` on the `document.body` with `$container`. This ensures the dialog is the last element in the DOM tree, which will help ensure the dialog gains the correct z-index, which we'll focus more on later. Finally, set `$state` to `open`.

```
this.$container = document.createElement(`div`);
const clone = template.content.cloneNode(true) as HTMLEle\
ment;
this.$container.classList.add(this.$variant);
this.$container.appendChild(clone);
document.body.appendChild(this.$container);

this.$state = "open";
```

If you open the Elements tab in Dev Tools, you should now observe a clone of the HTML template has been appended to the `document.body`. You just coded the core functionality of `DialogComponent`. You passed the value of HTML attributes using `observedAttributes` and `attributeChangedCallback`, then declared methods on the `class` that handles the target and opening a dialog. In the next section you'll stand up two new autonomous custom elements: `ModalComponent` and `TooltipComponent` that will handle some basic view behaviors for each dialog variant.

Modal and Tooltip

We've abstracted a single dialog's state to `DialogComponent`, which so far handles opening the dialog and based on the model passed through HTML attributes, displays the appropriate variant: modal or tooltip.

Modals in the application are classified as dialogs that take complete focus away from the current view. Modals typically appear in the center of the view, display a custom template, but can universally be cancelled via a button in the top right when the user clicks outside of the modal, or if the user presses the Escape key when the modal is in focus.

Tooltips are contextual dialogs that display in relation to content the user activated. Tooltips typically display beneath the content in

a small dialog that can hold contextual information, usually a short phrase or instruction for the user.

The first dialog we activated in the last section was wrapped by a generic `HTMlDivElement`. We'll change that soon. Using the `variant`, `DialogComponent` should display a `ModalComponent` or `TooltipComponent`. In this section, we'll introduce the boilerplate for these components and begin displaying the HTML templates in the context of each Shadow DOM template. That means `ModalComponent` and `TooltipComponent` should both be autonomous custom elements. We'll use template slots to inject the HTML templates into each component.

In this section we'll stand up the components, import them into `Dialog.stories.js` so they can be displayed in the story, integrate them with `DialogComponent` and observe how they behave before proceeding to the next section.

ModalComponent

To start development on `ModalComponent` create a new file in the dialog directory named `Modal.ts`.

```
touch packages/component/src/dialog/Modal.ts
```

Import the relevant `function` from @in/common for an autonomous custom element.

```
import { Component, attachShadow, html, css } from "@in/c\
ommon";
```

Export a `class` named `ModalComponent` that extends `HTMLElement`. In the `constructor` call `attachShadow`.

```
export class ModalComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this, {
      mode: 'open',
    });
  }
```

Use the `Component` decorator to decorate the `class`, passing in the `ElementMeta` with a selector that specifies the custom element should be registered with the tag name `in-modal`. Provide empty template strings for the `style` and `template` properties so you can begin to fill them in.

```
@Component({
  selector: 'in-modal',
  style: css``,
  template: html``,
})
```

The template for `ModalComponent` is as simple as it gets. Since we're levering HTML templates for each dialog, we simply have to pass the template to a slot.

```
<slot></slot>
```

After observing the styling for a modal in Figma we need to come up with a way `ModalComponent` can appear on top of the rest of the layout, while positioned at the center. This can be achieved by selecting the `:host` and ensuring the custom element is a block element with `display: block`. This will allow every `ModalComponent` to be reliably positioned and sized. To center the modal, give the element `position: fixed` and adjust the `top` and `left` to `50%` (relative to the `document.body` which roughly centers

the component. Use a CSS `transform` to affix the modal perfectly in the center of the view with a horizontal and vertical translation set to `-50%` (relative to the size of the modal). Give the modal a `z-index` set to `9000`. This is a relatively high index in the `z-index` layering over that ensures all other elements have an opportunity to properly layer, while dialogs can always display above all other content.

```
:host {
  display: block;
  position: fixed;
  top: 50%;
  left: 50%;
  transform: translateX(-50%) translateY(-50%);
  width: 50vw;
  z-index: 9000;
}
```

We want to also fix some potential issues that could happen with elements passing through the slot. Due to giving the modal a `fixed` position, direct descendants of the element won't display with the appropriate size that fills out the dialog. To fix this, select all slotted elements with the pseudo selector `::slotted(*)` and give them an `absolute` position, a `width` set to `100%` and `auto` height.

```
::slotted(*) {
  position: absolute;
  width: 100%;
  height: auto;
}
```

In Figma, it appears `ModalComponent` have a different box shadow compared to `CardComponent`, which appear inside the context of a modal. According to design, this is because dialogs have a much greater `z-index` so they get a different visual treatment. `box-shadow`

is typically the domain of `CardComponent` so override the default styling in that component using the `::slotted` pseudo selector once more. Pass `box-shadow` a CSS variable derived from the design stylesheet named `var(--shadow-modal)`.

```
::slotted(in-card) {
  box-shadow: var(--shadow-modal);
}
```

Once styling is complete for `ModalComponent`, head back over to `Dialog.stories.js` and import the `class`.

```
import { ModalComponent } from "./Modal";
```

Add `ModalComponent` to the exports in `Dialog.stories.js` as well so the Webpack config for Storybook can properly build the story.

You shouldn't be able to observe any changes yet in the story because you haven't integrated `ModalComponent` with `DialogComponent` yet. In the previous section, you listened for HTML attributes on `DialogComponent` and set the value of the `$variant` property, which can be equal to either `modal` or `tooltip`.

```
case 'variant':
    this.$variant = next;
break;
```

Update the `onOpen` method by deprecating the `div` that was there for test purposes, instead opting for the tag name of the variant.

```
this.$container = document.createElement(`in-${this.$vari\
ant}`);
```

You should now be able to observe a styled `ModalComponent` in Storybook when you press the button labeled "Open" in the Primary

dialog story. Keen observers will notice in the HTML template currently displayed that there is a button labeled "More Info". Upon clicking that button, another dialog opens. You're observing the `DialogComponent` in action. Each HTML template contains an instance of the `DialogComponent` custom element declared with the proper HTML attributes that specify the relation to a target.

Each modal displays a button in the top right that is intended to be used to close the modal upon user click. We'll deal with this feature in a bit, along with any other use cases that close the modal dialog.

In the second modal that appears above the first by virtue of being appended to the `document.body` with the same `z-index`, you can observe some boilerplate "Lorum ipsum" body copy, followed by a `(?)`. The intention is that anywhere a `(?)` appears, the user should be able to click and observe a tooltip with contextual information. Let's continue by standing up another autonomous custom element named `TooltipComponent`.

TooltipComponent

Make a new file in the `dialog` directory named `Tooltip.ts`.

```
touch packages/component/src/dialog/Tooltip.ts
```

Import `Component` and `attachShadow` from the `@in/common` package.

```
import { Component, attachShadow, html, css } from "@in/c\
ommon";
```

Export a `class` named `TooltipComponent` that extends `HTMLElement`, being sure to call `attachShadow` in the `constructor`.

```
export class TooltipComponent extends HTMLElement {
  constructor() {
    super();
    attachShadow(this, {
      mode: "open",
    });
  }
}
```

Declare the `Component` decorator in the same way you did the modal, although this time specify the tag name as `in-tooltip`.

```
@Component({
  selector: 'in-tooltip',
  style: css``,
  template: html``,
})
```

The template for `TooltipComponent` is the same as `ModalComponent`, a single template slot that is intended for any HTML template that displays a small amounft of information for the end-user.

```
<slot></slot>
```

Styling the `TooltipComponent` is very similar to `ModalComponent`; however, you need to omit position styles here because those will have to be dynamic to support the Acceptance Criteria that states "`TooltipComponent` should display a contextual tooltip just beneath an element the user triggered display". According to Figma, `TooltipComponent` should have a maximum width of `300px`. You'll need to apply the same fix for any slotted content so it properly fills its container. Looking once more at the Figma, it appears tooltips should display a different box shadow than modals or the default used by `CardComponent`. Use this opportunity to also override the `CardComponent` styles.

```
:host {
  display: block;
  position: fixed;
  min-width: 240px;
  max-width: 300px;
  z-index: 9000;
}
::slotted(*) {
  position: absolute;
  width: 100%;
  height: auto;
}
::slotted(in-card) {
  box-shadow: var(--shadow-tooltip);
}
```

Import `TooltipComponent` into `Dialog.stories.js`, making sure to also export the `class` in the same file.

```
import { TooltipComponent } from "./Tooltip";
```

Returning to Storybook, you should be able to click "Open", then "More Info" to display the second modal. Upon clicking "(?)" you'll observe an issue. The tooltip doesn't appear. Pretty anticlimactic, right? This is due to an issue with the implementation in `DialogComponent` we'll explore in depth in the next section.

Querying DOM Across Shadow Boundaries

There's a problem in the `DialogComponent` that causes the `TooltipComponent` not to appear when `DialogComponent` is declared in the context of the `ModalComponent` Shadow DOM. Since the

instances of `DialogComponent` is embedded in Shadow DOM, `DialogComponent` can't simply query for a HTML template using `document.querySelector`. The `querySelector` method doesn't cross the Shadow DOM boundaries imposed by the multiple instances of `ModalComponent`.

There is one way we could fix this. `ModalComponent` could be an autonomous custom element that doesn't use Shadow DOM. We could use other methods from the `@in/common` package so modals get the appropriate styling, like we have done with customized built-in elements in previous chapters. If we choose this approach, we lose the main benefit of Shadow DOM: the encapsulated style and template. Without Shadow DOM, `querySelector` would be able to function per normal.

Another approach that would allow us to retain Shadow DOM in `ModalComponent` is to code a new utility that can query for elements while piercing the Shadow boundary. For that, we'd need a `function` that can walk the tree of nodes that make up DOM and if an element has a `shadowRoot`, recursively call the `function` and walk that tree of nodes, trying to query for the element we need to select.

Luckily, a method exists on the `document` already that could help walk a tree of nodes. `document.createTreeWalker`[67] is a specialized method for setting up a `TreeWalker Object`. While holding a reference to the `TreeWalker`, you can use a `while` loop to iterate through every node in a tree of nodes, starting at the `document.body` or any other node, i.e. the `shadowRoot`. `createTreeWalker` doesn't by default cross the shadow boundary, so we'll have to code some custom logic to ensure that happens, but `TreeWalker` does provide an extremely performant means to walk a tree of nodes.

[67] https://developer.mozilla.org/en-US/docs/Web/API/Document/createTreeWalker

```
const walk = document.createTreeWalker(
  element ? element : (document.body as Element),
  NodeFilter.SHOW_ELEMENT,
  {
    acceptNode(node) {
      return NodeFilter.FILTER_ACCEPT;
    },
  },
  false
);
while (walk.nextNode()) {}
```

In this section, you'll move back to the library code in the `@in/common` package and code a new `function` named `query` which returns a recursive `function` that gets called every time the `TreeWalker` encounters a node that has a `shadowRoot`. The recursive `function` should get called until the last node in the tree has been parsed or a match is found via `querySelector`. `query` will be exported from the library and imported into the `DialogComponent` just like any other `function` from the `@in/common` package.

Make a new file named `query.ts` in the directory `packages/common/src/template`.

```
touch packages/common/src/template/query.ts
```

In `query.ts` export a new `function` named `query`. This function will work much like `querySelector`, in that the first argument takes the `selector`. Unlike `querySelector` which exists on the prototype of every `Element`, `query` is a "pure" `function`, so we'll need to pass the starting point for the query in the second argument named `root`. `query` returns an `Element`, the element we are selecting with the `selector`.

This algorithm needs to be recursive because there will be instances where the `TreeWalker` will reach a shadow boundary and stop, but

we want to run the algorithm again, starting at the `Shadow Root` and walking the tree of nodes from there, querying for the element that should match the `selector`. Return another `function` named `__query` that takes similar arguments. This time the second argument represents a match, which only happens once, but we need to pass it recursively and check if there's a match at the start of the `function`. The third argument is another `Element` which we'll use similarly to the `query function`, a starting point to query for the element. The arguments get flipped here because `element` isn't required in the initial call to `__query`. Declare the `__query function` here in the context of `query` and make sure it returns the `Element` to satisfy the type definitions for now.

```
export function query(selector: string, root?: HTMLElemen\
t): Element {
  function __query(
    selector: string,
    match: Element,
    element?: Element
  ): Element {
    return match;
  }
  return __query(selector, root);
}
```

The code we'll write goes inside of the recursive `__query function`. At the start of this `function` check if you can just use `querySelector` to find the element and set the value of `match` with what is returned by `querySelector`. If `element` is not `undefined` here, we'll want to skip to the next line. On the next line, use a conditional to check if `match` is not `undefined` and return `match`. This will short circuit the recursive `function` if an element is found.

```
if (!element && document.querySelector(selector)) {
  match = document.querySelector(selector);
}

if (match) {
  return match;
}
```

If `match` is `undefined`, we want to instantiate a new instance of `TreeWalker` by setting a new `const` named `walk` and passing all the arguments required by `document.createTreeWalker`. The first argument is the root node to start walking, which should be the `document.body` if `element` is `undefined`. The next argument optional for `createTreeWalker` is a way to filter types of nodes. We'll use the `NodeFilter.SHOW_ELEMENT` which filters only `Element` nodes and discards other nodes like `Comment` or `DocumentFragment`. The next argument is a filter for the nodes that pass the first filter, it tells when `TreeWalker` should accept a node. We need `FILTER_ACCEPT` here. Set the fourth argument to `false`, a boolean that indicates if an entire sub-tree should be discarded when a reference is also discarded. For more information about `TreeWalker`, view the documentation on MDN[68].

```
const walk = document.createTreeWalker(
  element ? element : (document.body as Element),
  NodeFilter.SHOW_ELEMENT,
  {
    acceptNode(node) {
      return NodeFilter.FILTER_ACCEPT;
    },
  },
  false
);
```

[68] https://developer.mozilla.org/en-US/docs/Web/API/Document/createTreeWalker

With a reference to the `TreeWalker` stored on `walk`, we can iterate through each `Element` as the tree is being walked with a `while` loop, calling the method `nextNode` on `walk`.

```
while (walk.nextNode()) {}
```

Inside the `while`, use an `if` statement to check if the `currentNode` can query for the `selector`. When truthy, set `match` to the result of the `querySelector`.

```
if ((walk.currentNode as Element).querySelector(selector)\
) {
  match = (walk.currentNode as Element).querySelector(sel\
ector);
}
```

Write another conditional that checks if the `currentNode` has a `shadowRoot` which `querySelector` can't pierce. Call `__query` recursively, passing the `selector` and `match` (which may still be `undefined`), and the `shadowRoot`.

```
if ((walk.currentNode as HTMLElement).shadowRoot) {
  __query(
    selector,
    match,
    (<unknown>(walk.currentNode as HTMLElement).shadowRoo\
t) as Element
  );
}
```

After you're done with the `while` loop, make sure the `__query function` still returns `match` at the very end. That's all there is to coding an algorithm that recursively walks a tree of nodes and queries for a single `Element`, but what if you wanted to replicate the functionality of `querySelectorAll`? Let's do just that.

In the same file declare another recursive implementation. This time wrap the recursive `function` named `__queryAll` with a `function` named `queryAll`. Both `function` take the same arguments as `query` and `__query` with one exception. `__queryAll` second argument should be an `Array` of `Element` because `queryAll` can return multiple matches.

```
export function queryAll(selector: string, root?: HTMLEle\
ment): Element[] {
  function __queryAll(
    selector: string,
    matches: Element[],
    element?: Element
  ): Element[] {
    return matches;
  }
  return __queryAll(selector, [], root ? root : null);
}
```

The start of `__queryAll` is very similar. Check that `element` is falsey and the `matches Array` is empty. Query using the selector from the `document` using `querySelectorAll` and concatenate the `Array` with the results of the query.

```
if (!element && !matches.length) {
  matches = matches.concat(Array.from(document.querySelec\
torAll(selector)));
}
```

The implementation of `TreeWalker` is exactly the same, although what happens on each loop of the `while` changes.

```
const walk = document.createTreeWalker(
  element ? element : (document.body as Element),
  NodeFilter.SHOW_ELEMENT,
  {
    acceptNode(node) {
      return NodeFilter.FILTER_ACCEPT;
    },
  },
  // @ts-ignore
  false
);
while (walk.nextNode()) {}
```

Check here if the `currentNode` has any matches to the `selector` using `querySelector`, followed by another `if` statement that checks if the `matches Array` doesn't contain the `currentNode`. It could because this algorithm walks down the tree. If both conditions are truthy, set `matches` to the `NodeList` returned by `querySelectorAll`, but use `Array.form` to convert the `NodeList` into an `Array` before concatenating the `matches`.

```
if ((walk.currentNode as Element).querySelector(selector)\
) {
  if (!matches.find((match) => match === walk.currentNode\
)) {
    matches = matches.concat(
      Array.from((walk.currentNode as Element).querySelec\
torAll(selector))
    );
  }
}
```

Still in the `while` loop, check if the `currentNode` has a `shadowRoot` and if so, recursively call `__queryAll` to walk that tree of nodes.

```
if ((walk.currentNode as HTMLElement).shadowRoot) {
  __queryAll(
    selector,
    matches,
    (<unknown>(walk.currentNode as HTMLElement).shadowRoo\
t) as Element
  );
}
```

After making sure `__queryAll` still returns `matches` at the end of the `function`, export both of the `functions` from `query.ts` if you haven't already. Open the `index.ts` file in `packages/common/src/template` and add the proper exports.

```
export { query, queryAll } from "./query";
```

Open `packages/common/index.ts` and export `query` and `queryAll` from there as well.

```
export {
  attachShadow,
  attachStyle,
  attachTemplate,
  html,
  css,
  query,
  queryAll,
} from "./src/template";
```

Build the `@in/common` package using lerna so you can pickup the changes in `DialogComponent`.

```
lerna exec --scope @in/common -- yarn build
```

If you have the Storybook server running, stop it momentarily, and restart the process.

```
yarn storybook
```

In `Dialog.ts` import the `query function`.

```
import { Component, attachShadow, query } from '@in/commo\
n';
```

Deprecate the call to `querySelector` in the `onOpen` method of `DialogComponent` and use `query` instead.

```
this.$target = query(this.$targetSelector);
```

At this point, you should be able to observe that an instance of `TooltipComponent` appears in the view when you click the (?) in the second modal found in the Primary dialog story. The tooltip still isn't positioned where it should be, just below the element you clicked, so let's fix that. Before, when calling `onOpen`, you declared the first argument of this method to be a `MouseEvent` because the method is bound to `addEventListener`.

```
  onOpen(ev: MouseEvent) {
```

We need a way to pass the position and size of the element that was clicked to contextually position the `TooltipComponent` below the element. `CustomEvent` works well in this scenario because we have direct access to a reference of the element the event should be dispatched to. In the `onOpen` method, after the `#container` has been appended to DOM, check if the `$variant` is equal to `tooltip` and if truthy, call `dispatchEvent` on the `$container` which is sure to be an instance of `TooltipComponent`. Pass `dispatchEvent` a `CustomEvent` named `position` and set a property named `rect` on the `detail` property to the `DOMRect` of the element the user clicked on, which is available via `ev.target`. For more information about `DOMReact`, visit MDN[69].

[69] https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect

```
if (this.$variant === "tooltip") {
  this.$container.dispatchEvent(
    new CustomEvent("position", {
      detail: {
        rect: (ev.target as Element).getBoundingClientRec\
t(),
      },
    })
  );
}
```

Open `Toolip.ts` and import `Listen` from `@in/common` so we can listen for the `position` event.

```
import { Component, attachShadow, Listen } from "@in/comm\
on";
```

Add a new method on `TooltipComponent` named `onPosition` and decorate the method with the `Listen` decorator, passing the decorator the name of the event: `position`. In the method, set a `const` named `rect` to `ev.detail.rect`.

```
@Listen('position')
onPosition(ev: CustomEvent) {
    const rect = ev.detail.rect;
}
```

We're going to use the information from the `DOMRect` to position the `ToolipComponent` properly below the "(?)" and to size the `CardComponent`. To style the `CardComponent`, we need an easy way to reference it. `CardComponent` is not a direct descendant of the `TooltipComponent` Shadow Root. The `CardComponent` is injected through a template slot. Set up another method on `TooltipComponent` that queries for the `CardComponent` via the

`assignedNodes` on the template slot. `CardComponent` should be the only child, so you can safely select the first item in the `NodeList`. Type define `$child` as an `Element`.

```
get $child(): Element {
    return this.shadowRoot.querySelector('slot').assigned\
Nodes()[0] as Element;
}
```

In the `onPosition` method, we're going to style the `TooltipComponent` position and size using two `DomRect`: one from the `CardComponent` which dictates the size of the `TooltipComponent` and the other from the `rect` passed from the element the user clicked on. We can use that `DomRect` to find the position of the `bottom` of the element and use that position to style the `top` of the `TooltipComponent`.

First, calculate the amount of displacement you'll need to center the tooltip left/right, which is equal to half the width of the `CardComponent`. Style the `top` of the `TooltipComponent` using the `rect.bottom` while adding `12` pixels to offset the tooltip vertically. Style the `left` using the `rect.left` minus the `leftDisplacement`, which has the effect of centering the tooltip. Style the `width` and `height` to the `CardComponent` `width` and `height`. Finally, ensure the element is visible by setting the `visibility`.

```
const leftDisplacement = this.$child.getBoundingClientRec\
t().width / 2;
this.style.top = `${rect.bottom + 12}px`;
this.style.left = `${rect.left - leftDisplacement}px`;
this.style.width = `${this.$child.getBoundingClientRect()\
.width}px`;
this.style.height = `${this.$child.getBoundingClientRect(\
).height}px`;
this.style.visibility = "visible";
```

With these styles in place, you should now be able to observe in Storybook that when clicking "(?)" in the second modal, the tooltip appears positioned underneath the cursor. This is partly due to the "(?)" being encapsulated by a `HTMLSpanElement`, which gives the `ev.target` a `DOMRect`.

Now you have considered all the use cases for opening dialogs while developing the `ModalComponent` and `DialogComponent` it's now time to think about how to close the dialogs in a user friendly way. In the next section you'll do just that.

DialogStack

In `DialogComponent`, you stored the state of a single dialog, but haven't yet found a way to orchestrate the rendering of several dialogs at once. You also haven't handled the "closed" state of a dialog, which is what happens when the user closes the dialog either by pressing the "Escape" key, the close button where available, or by clicking outside of the dialog.

Several problems could arise from the current implementation if we were to attempt to handle all these use cases. Nothing is currently tracking if a dialog is focused. If we add event listeners to each dialog we could end up with many different event listeners which could hurt performance, however unlikely given the number of dialogs in the current implementation. Perhaps it would be a good idea to delegate the focus event listener to a single place that also tracks the currently focused dialog.

Having all dialogs on the same `z-index` could also cause issues. We'd have to rely on the order of the elements in DOM which means a lot of DOM manipulation each time the order of dialogs changes. Suppose the user clicks on another dialog in the background. The user would expect that dialog to come to the front. In order to do that currently, we'd need to change the order of

DOM nodes appended to the `document.body`. Nothing can handle a change like that yet, but perhaps a more performant solution would be to set the `z-index` of each dialog in a well defined range of `z-index` reserved only for dialogs: 9000-9999.

To orchestrate the rendering of several dialogs, let's introduce a new `class` named `DialogStack`. The responsibility of `DialogStack` will be the following:

- Store order of dialogs rendered in the view
- Store template ids
- Store the currently focused dialog
- Store the previously focused dialog
- Delegate focus events for each dialog

`DialogStack` will require some integration with `DialogComponent`. When `DialogComponent` is instantiated and changes state, the component should update `DialogStack`. `DialogStack` would be a singleton and requires global scope to operate. For the purposes of this exercise, we'll instantiate `DialogStack` on the `window` so it's accessible globally.

To begin development on `DialogStack`, make a new file named `DialogStack.ts` in the same directory where you've been working on all the dialogs.

DialogStack

```
1 touch packages/component/src/dialog/DialogStack.ts
```

In `DialogStack.ts` export a new `class` named `DialogStack` and declare a `constructor`. Add two properties (`stack` and `templateId`) we know are required if `DialogStack` is supposed to store template ids and the order of dialogs rendered in the view.

```
export class DialogStack {
  stack: Element[] = [];
  templateId: string[] = [];
  constructor() {}
}
```

Let's first handle the registration and removal of template ids. Add a method named `registerTemplate` that will eventually be called from the context of `DialogComponent`. The only argument for the new `method` is a `string` named `id`. If the `id` is not found in the `templateId Array`, add the `id` to the `Array`.

```
registerTemplate(id: string) {
    if (!this.templateId.find((templateId) => templateId \
=== id)) {
        this.templateId.push(id);
    }
}
```

Declare another method that finds the index of an `id` in the `templateId Array`. You'll need this `index` to remove an `id` from the `Array`.

```
findTemplateIndex(id: string) {
    return this.templateId.findIndex((templateId) => temp\
lateId === id);
}
```

Make a method on the `DialogStack class` named `removeTemplate`. Using the only argument `id`, call `findTemplateIndex`. Splice the `templateId` at the `index` of the `id` if it exists. This has the effect of removing the `id` from the `Array`.

```
removeTemplate(id: string) {
    const templateIndex = this.findTemplateIndex(id);
    if (templateIndex) {
        this.templateId.splice(templateIndex, 1);
    }
}
```

With `templateId` state covered, let's turn our attention to managing the `z-index` of each dialog in the `stack Array`. We haven't populated the `Array` yet, but supposing we did, the dialog at the last `index` in the `Array` would be the dialog that should be layered on top of every other. Set a `const` named `base` to `9000`, the first (and lowest) `z-index`. Iterate through the `stack` with `forEach`, setting the `zIndex` of each dialog, incrementing the `z-index` by the `index` of the dialog in the `Array`.

```
setZIndices() {
    const base = 9000;
    this.stack.forEach((element, index) => {
        (element as HTMLElement).style.zIndex = (base + i\
ndex * 10).toString();
    });
}
```

Make another method that adds a dialog to the `stack Array`. The argument for this method is typed `Element` to keep it generic, but it might as well be `ModalComponent | TooltipComponent`. Call `push` on the `Array.prototype` to add the dialog to the `stack Array`. `DialogComponent` currently handles rendering the dialog, but in a member you'll deprecate that functionality from `DialogComponent` because it makes a lot more sense to add the call to `appendChild` here. After the dialog is appended to the `document.body`, call `setZIndexes`.

```
pushDialog(element: Element) {
    this.stack.push(element);
    document.body.appendChild(element);
    this.setZIndices();
}
```

Integrate DialogStack with DialogComponent

Import `DialogStack` into `Dialog.ts`.

```
import { DialogStack } from "./DialogStack";
```

Declare a new type interface for the `DialogStack`. This `class` should be global, so we'll store it on the `window` for now.

```
declare global {
  interface Window {
    __dialogStack: DialogStack;
  }
}
```

Import `DialogStack` into `Dialog.stories.js` so it can be used in the context of the story. You should also export the `class` from the file.

```
import { DialogStack } from "./DialogStack";
```

Instantiate a new instance of `DialogStack` on the `window`. By setting a property named `__dialogStack` on `window` to an instance of `DialogStack`, we can reference this instance from `DialogComponent`.

```
window.__dialogStack = new DialogStack();
```

`DialogComponent` requires some modification to integrate with `DialogStack`. Update the `attributeChangedCallback` with a call to `registerTemplate` on `window.__dialogStack`, but only if a template is not found by calling `findTemplateIndex`.

```
  attributeChangedCallback(name, prev, next) {
    switch (name) {
      case 'target':
        if (window.__dialogStack.findTemplateIndex(next) \
=== -1) {
          window.__dialogStack.registerTemplate(next);
          setTimeout(() => this.setTarget(next), 1);
        }
        break;
```

In the `onOpen` method, remove the call to `document.body.appendChild` and update the method with a call to `pushDialog` on `window.__dialogStack`. The new API includes the ability to append a dialog to the `document.body`.

```
    this.$state = 'open';
    window.__dialogStack.pushDialog(this.$container);
}
```

Handling Closed State

`DialogStack` has the ability to add dialogs and sort their layering order, but we haven't included any logic to handle removing dialogs from DOM. When the user closes a dialog, the dialog should be removed. Dialogs are stored on the `DialogStack` `class` in an `Array`. We would need to get the index of the dialog

via the `Array.prototype.findIndex` method so we can later call `Array.prototype.splice` to remove the `Element` from the `Array`. Declare a new method on `DialogStack` that returns the index of an `Element` in `stack Array`.

```
findDialogIndex(element: Element): number {
  return this.stack.findIndex((elem) => elem === element);
}
```

In addition to removing the `Element` from state, we also need to query for the `DialogComponent`, which controls the state of a single dialog. Since instances of `DialogComponent` are not necessarily direct children of the `document.body`, we can't use `querySelector`. Use cases like this one is what `query` was made for, so we'll import that `function`.

```
import { query } from "@in/common";
```

Declare another method on `DialogStack` called `removeDialog` and pass the method a single argument, an `Element`. Every dialog has a HTML attribute that contains an id you can use to query for the `DialogComponent`. `DialogComponent` use the `data-target-id` HTML data attribute for this id, while each dialog uses the `data-dialog-id` HTML data attribute.

```
removeDialog(element: Element) {
  const id = element.getAttribute('data-dialog-id');
  const target = query(`[data-target-id="${id}"`);
}
```

Dispatch a `CustomEvent` to the `DialogComponent` with the name `close`. Maintain the current state of dialogs in `DialogStack` by splicing the `Array`, effectively removing the dialog from state. Remove the `Element` from DOM using `removeChild` and call `SetZIndices` in case any other dialogs need to be reshuffled.

```
removeDialog(element: Element) {
  const id = element.getAttribute('data-dialog-id');
  const target = query(`[data-target-id="${id}"`);
  if (target) {
    target.dispatchEvent(new CustomEvent('close'));
  }
  this.stack.splice(this.findDialogIndex(element), 1);
  document.body.removeChild(element);
  this.setZIndices();
}
```

Update `DialogComponent` with a new method that handles changing the `$state` from `open` to `close`, being sure to call `removeDialog` and `removeTemplate` on the `DialogStack`.

```
  onClose() {
    window.__dialogStack.removeDialog(this.$container);
    window.__dialogStack.removeTemplate(this.$targetSelec\
tor);
    this.$state = 'close';
  }
```

In the `onOpen` method, query for a "close" button that should have the CSS class name `.dialog-close-button` and if there is an instance of the button, use `addEventListener` to handle the `click` event on the element. Call `onClose` in the event handler callback.

```
const closeButton = this.$container.querySelector(".dialo\
g-close-button");

if (closeButton) {
  closeButton.addEventListener("click", (ev) => {
    this.onClose();
  });
}
```

Whenever `DialogComponent` is removed from DOM, also remove the template from `DialogStack` state.

```
disconnectedCallback() {
    window.__dialogStack.removeTemplate(this.$targetSelec\
tor);
}
```

Handling Focus

To handle the use case when a user clicks outside of a dialog, which should cause it to close, `DialogStack` needs a way to handle `focus` events. Delegating the event in `DialogStack` seems like a good idea so each `DialogCompoment` doesn't attach a duplicate event listener. Add two new properties to `DialogStack`: `last` and `focused` that will store the last focused and currently focused `Element`.

```
last: Element;
focused: Element;
```

Add another method named `addListeners` that will eventually be called in the `constructor` to attach the event listener to the `document.body`. Pass the event listener callback a new method that hasn't been declared yet on `DialogStack`: `onFocus`.

```
addListeners() {
    document.body.addEventListener('click', this.onFocus.\
bind(this));
}
```

The click event is on `document.body` because we're using event delegation. In the `onFocus` callback, using the `ev.target` (the `Element` the user clicked), call `closest` and query for the nearest dialog. `closest` is a method on every `Element` that walks up the DOM tree to find the closest `Element` that matches the selector. More info about closest on MDN[70].

```
 onFocus(ev: MouseEvent) {
     const closest = (ev.target as Element).closest('[dat\
a-dialog-id]');
 }
```

Whenever a tooltip is currently `focused`, we can assume that the user wants to dismiss the tooltip if they didn't click on the tooltip. Had the user clicked on the tooltip, `closest` would return something, but without a match, `closest` returns `null`. If the tooltip is the currently `focused Element`, call `removeDialog`, passing in the reference to the `Element`.

```
if (closest == null && this.focused && this.focused.tagNa\
me === "IN-TOOLTIP") {
  this.removeDialog(this.focused);
}
```

To maintain the state of the previous and current focused `Element`, make a new method named `focusDialog` that sets `last` and `focused`

[70] https://developer.mozilla.org/en-US/docs/Web/API/Element/closest

```
focusDialog(element: Element) {
    this.last = this.focused;
    this.focused = element;
}
```

Call `focusDialog` in `pushDialog` to focus the dialog that was just appended to DOM.

```
this.focusDialog(element);
```

In `removeDialog`, focus the dialog that is the last item in the `stack Array`.

```
this.focusDialog(this.stack[this.stack.length - 1]);
```

Add the call to `addListeners` in the `constructor` to handle the event listener.

```
constructor() {
    this.addListeners();
}
```

After manually testing out the changes to `DialogStack` and `DialogComponent` in Storybook, you observe that modals correctly close and focus based on user interaction. You are now ready to write some automated tests for the dialogs.

Testing Dialogs

In this section, you'll write some integration tests for the dialog with Cypress, using Storybook as the testbed. Integration tests are crucial for the dialogs, which at this point have several moving parts: `DialogComponent`, `DialogStack`, `ModalComponent` and `TooltipComponent`. You'll test the following:

- Should display a dialog
- Should close a dialog
- Should close and open dialog again
- Should open another dialog
- Should open and close tooltip
- Should open and close all dialogs

After coding the integration tests and observing all the tests pass with Cypress, you'll follow up with some snapshot tests.

Integration Tests

In each of the integration tests, you'll use the `Primary` dialog story from this chapter as a test bed.

Make a new file in the `cypress/integration` directory called `Dialog.spec.js`.

```
touch cypress/integration/Dialog.spec.js
```

Add a `describe` block to the test file and use a `beforeEach` to mitigate any issues with testing the components in Storybook.

```
describe("Dialog Component", () => {
  beforeEach(() => {
    cy.wait(1);
  });
});
```

Should Display A Dialog

Test if a dialog is displayed by default. A dialog should automatically be displayed in the `Primary` story because the `args` are configured that way.

With Cypress, call `cy.visit` with the query params necessary to navigate to the `Primary` dialog story in Storybook. Test if an element with tag name `in-modal` exists in the view.

```
it("should display a dialog", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root").get("in-modal").should("exist");
});
```

Should Close A Dialog

Set up another test that visits the same path in Storybook. This time use `find` to drill into the slot that contains the buttons in a modal. Call `click` on the button. When the button is clicked, the modal should be removed from DOM. Test this with the assertion `should("not.exist")`.

```
it("should close a dialog", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root")
    .get("in-modal")
    .find('[slot="action"]')
    .find("button")
    .click();
  cy.get("#root").get("in-modal").should("not.exist");
});
```

Should Open and Close A Dialog

The next test builds off the last, although this time, instead of checking that the `ModalComponent` doesn't exist after it's closed, direct Cypress to click the "Open" button again and test if the `ModalComponent` exists.

```
it("should close and open dialog again", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root")
    .get("in-modal")
    .find('[slot="action"]')
    .find("button")
    .click();
  cy.get("#root").get("button").click();
  cy.get("#root").get("in-modal").should("exist");
});
```

Should Open Another Dialog

Direct Cypress to click the "More Info" button in the footer of `ModalComppnent` and write an assert that a second `ModalComponent` exists.

```
it("should open another dialog", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root")
    .get("in-modal")
    .find('[slot="footer"]')
    .find("button")
    .click();
  cy.get("#root").get("in-modal").eq(1).should("exist");
});
```

Should Open And Close Tooltip

This next test is a bit more verbose. Building off the last few tests, open the second modal with Cypress.

```
it("should open and close tooltip", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root")
    .get("in-modal")
    .find('[slot="footer"]')
    .find("button")
    .click();
});
```

Target the "(?)" `HTMLSpanElement` and call `click`, opening the tooltip. Follow that by directing `Cypress` to click on the `#root`, whitespace surrounding the `ModalComponent`. Assert that the `TooltipComponent` should not exist.

```
cy.get("#root")
  .get("in-modal")
  .eq(1)
  .find('[slot="content"]')
  .find('[data-dialog-id="tooltip-help-target"]')
  .click();
cy.get("#root").click();
cy.get("#root").get("in-tooltip").should("not.exist");
```

Should Open And Close All Dialogs

The final integration test opens and closes all the dialogs to test that every instance of `ModalComponent` and `TooltipComponent` can be opened and closed.

Start with a very similar command that opens the second `ModalComponent`.

```
it("should open and close all dialogs", () => {
  cy.visit("iframe.html?id=components-dialog--primary");
  cy.get("#root")
    .get("in-modal")
    .find('[slot="footer"]')
    .find("button")
    .click();
});
```

Direct Cypress to click on the "(?)" in the second modal, which opens the tooltip. Cypress should then click the `#root`, which should close the tooltip. Tell Cypress to click the "close" button in the footer of the open `ModalComponent`. This should close the modal.

```
cy.get("#root")
  .get("in-modal")
  .eq(1)
  .find('[slot="content"]')
  .find('[data-dialog-id="tooltip-help-target"]')
  .click();
cy.get("#root").click();
cy.get("#root")
  .get("in-modal")
  .eq(1)
  .find('[slot="footer"]')
  .find(".dialog-close-button")
  .click();
```

Direct Cypress to click on the "close" button of the first modal, followed by assertions that neither `ModalComponent` or `TooltipComponent` exist.

```
cy.get("#root").get("in-modal").find('[slot="action"]').f\
ind("button").click();
cy.get("#root").get("in-modal").should("not.exist");
cy.get("#root").get("in-tooltip").should("not.exist");
```

Snapshots

With Storybook still running, generate snapshots for the component.

```
yarn test:snapshot:ci
```

When snapshots have been generated for the Dialog story, add them to your commit. Commit your changes in git to complete this section. After you've completed integration and snapshot tests for `DialogComponent` let's review the content of Chapter Six.

Summary

Dialogs are a common user interface component in enterprise-grade applications. In this chapter you developed two types of dialogs: modals and tooltips. By abstracting each type to autonomous custom elements, you were able to isolate the view (`ModalComponent` and `TooltipComponent`) from another autonomous custom element that acted like a controller in the traditional sense of MVC. `DialogComponent` handled the state of a single dialog, delivering a model through HTML attributes to a dynamically generated instance of a dialog.

While giving engineers an interface to declare dialogs alongside the elements that trigger their display in DOM, you ran into an issue where Shadow DOM could obscure the elements from

being queried with `querySelector`. The `query function` you wrote traverses Shadow boundaries, querying for DOM nodes with a selector similar to `querySelector`.

After observing how several dialogs worked together, you built `DialogStack` which handled the state of many dialogs. You also delegated focus events to `DialogStack`, a singleton instantiated on the `window`.

By creating a separation of concerns between several components, hopefully you gained an appreciation of how to architect user interfaces in a sustainable and intuitive fashion. This is the last chapter in Part Two where you'll participate in development of complex custom elements. In later chapters, the development of the Header, Content, Footer layout should hopefully feel like a cakewalk compared to the development of the dialogs and table. Before developing custom elements in the context of an application, you'll learn best practices for distributing a UI library of custom elements in the next chapter.

Distributing Custom Element Libraries

Chapter Seven

UI Libraries

Design and Engineering

Cooperation between design and engineering in large enterprise organizations is crucial to ensure scalability and maintainability for a codebase. That cooperation often happens at the level of a UI library, a collection of user interfaces that handle repeatable views in an application. Figma connects everyone involved in the design process, giving designers a venue to publish UI designs and other stakeholders the ability to comment and interact with the design.

Design teams often follow the principals of atomic design, a philosophy popularized by designer and developer Brad Frost[71]. Atomic design[72] prescribes an architecture that begins at the smallest units: atoms, and scales into molecules and organisms. As this happens, the more complex the user interface becomes. Organisms are comprised of molecules, molecules are comprised of atoms. You've experienced this design pattern while developing the many user interfaces in this book. CSS variables that represent design tokens serve as atoms. `TextInputComponent`, `CardComponent` could be considered molecules, while `TableComponent` or `DialogComponent` are

[71] https://bradfrost.com
[72] https://shop.bradfrost.com

organisms. Understanding the atomic design pattern is paramount when designing and developing a UI library.

When realized in the context of a UI library, atomic design promotes a separation of concerns between UI components when components are built with a strict separation of concerns. If the user interfaces are built once and distributed across an organization in a UI library, it promotes consistency and uniformity across a product. UI libraries can reduce maintenance overhead. Unnecessary churn could happen if teams were to code the same user interfaces in many different ways. UI libraries built with JavaScript frameworks and libraries have the cost of dependency maintenance, an issue that is substantially reduced UI libraries are built entirely with browser specifications.

UI libraries can be developed in such a way that allows engineers to move fast and build features. Exporting discrete units of functionality promotes reuse. Distributing components that handle complex user interaction patterns is much more difficult. Suppose a team wanted to use `TableComponent` but needed the additional feature of selecting multiple rows. There isn't a mechanism in the library yet for a user interface that allows toggling something, like a checkbox. If a `CheckboxComponent` were already developed, adding one to each row of `TableComponent` would be tricky to avoid breaking changes, the feature would have to be opted in, probably by setting a HTML attribute. Making components like `TableComponent` composable is difficult in any library, but possible. You've already abstracted each row and cell to separate components. The `TrComponent` and `TdComponent` `class` could be extended to enhance them with other functionality. This example is just one of many problems that can arise because of how components are architected; however, by exporting the discrete units of functionality that comprise complex components, you can retain design characteristics and user experience behaviors while giving teams the opportunity to construct custom user experiences.

Continuous Integration

How UI libraries are integrated and distributed in an enterprise organization's development pipeline are as critical to their success as how they are architected. Continuous integration ensures that code which can't be properly built or tested isn't being introduced into the main branch of a repository. Although integration depends largely on the tooling that is available, in this chapter, you'll learn how to set up a continuous integration pipeline with Github Actions. By implementing a package format that teams can easily install and import, you'll implement best practices for distributing a UI library of Web Components.

In this chapter, you'll learn how to validate a pull request in Github by running integration and snapshot tests via Github Actions. Cypress and Storyshots both require Storybook to be properly built, so you'll learn how to run Storybook with Github Actions. When an engineer creates a pull request in Github, it's also a good idea to check the build is successful, otherwise the project cannot be properly built before it's deployed. Typically, deployment of any npm packages happens when a PR meets some criteria after being merged to the main branch. You'll set up everything in Github Actions leading up to that point, while also deploying Storybook to Github Pages to observe the current state of the components after merging to the main branch.

Distribution

It should be as easy as possible for teams to install and start using the components. JavaScript packages are usually distributed via npm, the node package manager. npm packages use the `package.json` file for configuring the package. By implementing a predictable package format that is easily interpreted by build tools like Webpack, Rollup, snowpack or esbuild, it ensures the components in a UI library are reused. These build tools interpret

the `package.json` in order to bundle the JavaScript, using the configuration therein to locate the entry point(s) of the JavaScript that is exported from the package.

Components in the UI library are strictly typed with TypeScript, although browsers cannot interpret TypeScript, only JavaScript. TypeScript also allows you to develop with specifications only available in future iterations of the ECMAScript specification that haven't been adopted in browsers yet, like the decorator pattern you've used when developing the components. TypeScript can be configured to compile down to a version of the ECMAScript spec that is compatible with the browsers you intend to support. To ensure support across a great range of browsers, you can compile the library to ES2015, a version of the ECMAScript specification that is found in all major evergreen browsers (Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge). The module pattern that wraps the code can also be configured with TypeScript. ES2015 modules can be interpreted by build tools and in the browser. Internet Explorer 11 was the last major browser to not implement ES2015. If you require support for IE11, additional steps must be taken to compile the code down to ES5, an earlier version of ECMAScript, and commonjs modules, which can be interpreted by IE11 browser. We're not going to explore configuration for IE11 in this book since Microsoft has signaled a deprecation timeline for the browser. In this chapter, you'll learn how to distribute a UI library that works in evergreen browsers.

Documentation

Documentation is an important part of the distribution strategy for a UI library. Giving engineers access to clear, concise documentation for each component allows them to easily understand how to integrate the component into their application. Luckily for us, Storybook provides exceptional tooling for documenting UI components. The Custom Elements Manifest Analyzer provides a

means to statically analyze the `class` definitions of each component. The Custom Elements Manifest is a JSON file located in the root of the repository. Storybook references the Custom Elements Manifest to configure documentation for each component. While parts of the components are self-documenting (their methods are analyzable), Custom Elements Manifest will also analyze any JSDoc comments attached to each `class`. In this chapter, you'll write inline documentation with JSDoc for each component that describes each component's tag name, public methods, and HTML attributes that are exposed. These comments, while providing inline documentation for engineers working on the UI library, will also become visible in Storybook in a well formatted set of documentation.

In this chapter, you're going to take a drastic turn in development. Instead of developing a component, you will mainly write configuration and code scripts that support build tools such as Lerna, TypeScript, Rollup, and Storybook. By the end of the chapter you should have a stable build for the `@in/ui package`, a UI library filled with the components you developed in previous chapters. You'll use the library package like any other npm package in Part Three, when you'll develop an application using the components found in the `@in/ui` package.

What you will build

- Bundle a UI library with Rollup
- Document each component with JSDoc
- Generate a Custom Elements Manifest
- Configuration for Github Actions

What you will learn

- Best practices for distributing a UI library

- How to bundle UI components with Rollup
- How to make a UI library treeshakable
- How to bake the process into CI with Github Actions
- How to work more efficiently in a monorepo
- How to maintain a separation of concerns with lerna

Acceptance criteria

- All components and type definitions are exported from `@in/ui` package
- All components are documented with JSDoc
- Custom Elements Manifest is generated in the root directory
- When an engineer makes a PR in the repository, Github Actions builds and tests the project
- When an engineer merges a PR to the main branch, Github Actions also builds Storybook for Github Pages

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-7 from where you left off at the end of chapter 6.

```
git checkout -b chapter-7
```

Distributing Custom Element UI Libraries

In this section, you'll prepare the UI library for distribution by optimizing the code using techniques that promote treeshaking and fast build times. You'll format the package so the directory structure is machine readable by TypeScript, which should statically analyze the package when it is a dependency of another project. Developers should be able to inspect the type definitions of each component they are importing into their project from the package. By the end of this section, you will have a library package that is compatible with build tools like Webpack, Rollup, snowpack, and esbuild.

Named Exports

The first thing we'll do is refactor how the components are exported from each directory in `packages/component/src`. So far, you only had to export anything from a file that was needed in the context of a story file, but now we have to consider how components should be exported for distribution in the UI library.

One strategy to implement here is the inclusion of a new file in each component directory: `index.ts`. The purpose of this file is to expose anything that should become part of the library, any public APIs. A tool at our disposal for exporting everything from a file that uses the `export` keyword is `export *`. However, this practice causes a bottleneck for build tools that prevents them from "treeshaking" the library, removing unused parts of the library that were not directly imported into another project. Named ES2015 exports are necessary to ensure a library is treeshakable. Named exports allow build tools to statically analyze dependencies far better, enabling them to treeshake ununsed parts of the library.

In each component directory, make a new file named `index.ts`. Observe in each component file what is currently exported using the

`export` keyword and export the same exports in the `index.ts`. After we walk through each component, we'll create another `index.ts` in the root directory of the `@in/ui package` that exports everything for the library using named ES2015 exports.

Make a new file in `packages/component/src/button` named `index.ts`.

```
touch packages/component/src/button/index.ts
```

After observing what is exported in `Button.ts`, add another named export in `index.ts` for `ButtonComponent`.

```
export { ButtonComponent } from "./Button";
```

Create another `index.ts` file in the card directory.

```
touch packages/component/src/card/index.ts
```

Export `CardComponent` from the new `index.ts` file.

```
export { CardComponent } from "./Card";
```

Make an `index.ts` file in the dialog directory.

```
touch packages/component/src/dialog/index.ts
```

Export all the necessary dialog components from the `index.ts`.

```
export { DialogStack } from "./DialogStack";
export { DialogComponent } from "./Dialog";
export { ModalComponent } from "./Modal";
export { TooltipComponent } from "./Tooltip";
```

Make an `index.ts` file in the input directory.

```
touch packages/component/src/input/index.ts
```

Export the necessary parts for anyone to use `TextInputComponent`.

```
export { validate, Validator } from "./validator";
export { TextInputComponent } from "./TextInput";
```

Make an `index.ts` file in the table directory.

```
touch packages/component/src/table/index.ts
```

Export the necessary parts of the table.

```
export { TableComponent, Column, ColumnData } from "./Tab\
le";
export { TableCardComponent } from "./TableCard";
export { TdComponent } from "./Td";
export { TrComponent } from "./Tr";
```

Export everything from `index.ts` in the `packages/component` directory.

```
export { ButtonComponent } from "./src/button";
export { CardComponent } from "./src/card";
export {
  DialogComponent,
  DialogStack,
  ModalComponent,
  TooltipComponent,
} from "./src/dialog";
export { TextInputComponent, validate, Validator } from "\
./src/input";
export {
  Column,
  ColumnData,
  TableComponent,
  TableCardComponent,
  TdComponent,
  TrComponent,
} from "./src/table";
```

By exporting the components with named 2015 exports, you've made the entire library treeshakable. If an engineer only wanted to import `CardComponent` into their project, they can do so without bundling the rest of the library. This would not have been possible necessarily with `export *`. Now that you're done formatting the source code for distribution, let's consider how the package will look if we build.

Build Optimizations

Building the library package for distribution means transpiling TypeScript down to a specific version of ECMAScript that is compatible with the browsers you have to support. You'll use the TypeScript compiler for this purpose. After the code is transpiled, another optimization we can provide to anyone using this package is bundling the entire library in one file that exports everything in

ES2015 modules. This pattern is called a FESM (Flat ECMAScript Module). By exporting a FESM for bundlers like Webpack, you can optimize how the library is built. Since the bundler only has one file to statically analyze as opposed to many, it can reduce build times. Rollup is an excellent tool for bundling a UI library into a FESM, so we'll use it.

The necessary build files have been supplied for you in `packages/component`. Let's observe what's happening in each file. Open `packages/component/rollup.config.js`. In this file, Rollup is configured to use TypeScript and resolve dependencies from npm. When rollup builds the project, it outputs a file in `packages/component/dist/fesm2015/index.js` formatted as a FESM. Rollup also provides sourcemaps in the same directory, which can greatly enhance developer experience in some use cases like when inspecting code in Google Chrome Dev Tools.

```
import typescript from "@rollup/plugin-typescript";
import nodeResolve from "@rollup/plugin-node-resolve";

export default [
  {
    input: "./index.ts",
    plugins: [nodeResolve(), typescript()],
    onwarn: (warning, next) => {
      if (warning.code === "THIS_IS_UNDEFINED") return;
      next(warning);
    },
    output: {
      file: "./dist/fesm2015/index.js",
      format: "esm",
      sourcemap: true,
    },
  },
];
```

As part of the rollup build, type definitions are outputted in `dist/fesm2015` but we want the type definitions to be outputted in the root directory of the package. To do this, we'll need to run the TypeScript compiler again and provide the following configuration in a `tsconfig.json` that tells the compiler to only emit type definitions in `dist/types`.

```
{
  "extends": "./../../tsconfig.json",
  "compilerOptions": {
        "declaration": true,
        "declarationDir": "dist/types",
        "emitDeclarationOnly": true,
        "target": "es2015"
  },
  "include": [
        "./**/*.ts"
  ],
  "exclude": [
        "./**/*.stories.js",
        "./dist"
  ]
}
```

Language Support

TypeScript has the ability to compile down to a specific version of the ECMAScript specification, which has been updated yearly with new features since 2015. Which version of the ECMAScript specification you compile depends on the end consumer. Decisions surrounding which browsers to support in an enterprise organization are largely guided by user metrics. Browsers support varying parts of the latest ECMAScript specifications, but we can reliably compile down to ES2015 because all evergreen browsers have implemented this version of the specification. Depending on how

may browsers you want to support, you maybe able to compile to a newer version of the ECMAScript protocol.

Other tools like Babel solve this problem differently. Relying on `browserslist` that specifies which browsers are supported, babel transpiles a codebase to a version of ECMAScript that fits the browser support profile. This sounds like an attractive solution, although possibly not the most optimal because another tool is added to the build pipeline. For that reason, we'll just output the lowest common denominator that works in all evergreen browsers: ES2015.

Build

After the package is transpiled with TypeScript and bundled with Rollup, the build script also copies the `package.json` file to the `dist` directory. This is to ensure paths configured in the `package.json` resolve correctly when the root directory is the `dist` directory. Inspect the build scripts in the `package.json`.

```
"scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "clean": "rimraf dist node_modules",
        "build": "yarn clean && make-dir dist && rollup -c rollu\
p.config.js && tsc -p tsconfig.json",
        "prepublishOnly": "copyfiles package.json dist"
  },
```

Package Format

To distribute a npm package, the root directory must include a `package.json`. This file specifies the name of the package (`@in/ui`), the version (`0.0.1`), information about licensing (`"license": "MIT",`) and the package's dependencies. Any package that is

imported into the UI library and referenced in source code should be declared in `dependencies`, while any dependency required for development should be declared in `devDependencies`. `@in/ui` only has one dependency: `@in/common`, the micro-library that exports decorators for component development. We could opt to include the micro-library with the UI library or allow downstream bundlers to resolve the dependency and bundle `@in/common` accordingly.

Bundlers will need some more configuration in the `package.json` to interpret the library package. Let's add the necessary configuration.

The `package.json` should include an entry that specifies the package exports ES2015 modules. Add the `type` property to the `package.json` and set the value to `module`.

```
"type": "module",
```

To provide the entry points of the UI library for the latest bundlers, declare the `exports`. The paths declared in this `Object` will be referenced by Webpack during development in the monorepo.

```
"exports": {
        "./package.json": {
          "default": "./package.json"
        },
        ".": {
          "types": "./dist/types/index.d.ts",
    "es2015": "./dist/fesm2015/index.js",
    "default": "./dist/fesm2015/index.js"
        }
}
```

For legacy bundlers, use the prior implementation which specified the entry point on the `main` field and the type definitions entry point on the `types` field.

```
"main": "dist/fesm2015/index.js",
"types": "dist/types/index.d.ts",
```

If we omit `dist` from these fields it may cause problems in the monorepo. Lerna and Storybook depend on the `main` and `types` fields in the `package.json` to map each package in the monorepo for development. In Part Three, when you import components from `@in/ui` into the application, lerna will not be able to function properly if the `main` and `types` fields do not point to the correct files in the dist directory. To correct this behavior, while also providing a clean `package.json` for distribution, we'll process the the `package.json` file prior to publishing to npm.

Make sure the `main` and `types` fields include `dist` in the path.

```
"main": "dist/fesm2015/index.js",
"types": "dist/types/index.d.ts",
```

The `prepublishOnly` script in the `package.json` found at `packages/component/package.json` runs prior to publishing to npm. The script currently copies the `package.json` to `dist`. We have an opportunity here to insert more custom logic that processes the file.

```
"prepublishOnly": "copyfiles package.json dist"
```

To make sure the `package.json` is formatted correctly for distribution, we'll need some way to process the file and output the edited file in the `dist` directory. Node includes tools in `fs` that allow scripts to interact with the local filesystem. The primary reason we need to edit the `package.json` prior to distribution is due to how it's formatted for use in the monorepo. Paths in the `packages/component/package.json` point to the `dist/` directory so other packages in the monorepo can source the built package during

development. When the library is built in dist, we need a script that edits those paths relative to the files in the `dist` directory instead.

The `function` `readFile` and `writeFile` can help us with this task. When referencing file paths in node scripts it's helpful to wrap the string in a `function` exported from `path` named `resolve`. This ensures the path will resolve correctly on any OS filesystem. Make a new file named `prepublish.js` in `packages/component`. Open `prepublish.js` in your IDE and import the necessary `function` from `fs` and `path`.

```
1  import { readFile, writeFile } from "fs";
2  import { resolve } from "path";
```

Call `readFile`, passing in the `package.json` that was copied to `dist` into the first argument. The second argument of `readFile` specifies the format the read should be reach, which is `utf-8`. The third argument is a `function` callback where the contents of the file are passed.

```
1  readFile(resolve("./dist/package.json"), "utf8", (err, js\
2  on) => {});
```

Parse the JSON with `JSON.parse` and begin editing the fields for the `package.json` that is distributed with the library package. Change the paths in the `main` and `types` fields in relation to the the `package.json` in `dist`.

```
readFile(resolve("./dist/package.json"), "utf8", (err, js\
on) => {
  const pkg = JSON.parse(json);
  pkg.main = "fesm2015/index.js";
  pkg.types = "types/index.d.ts";
});
```

The `scripts Object` is unnecessary for distribution. it only includes scripts that builds the library, so it can be omitted.

```
delete pkg.scripts;
```

Call `writeFile`, passing in the path to the `package.json` in the first argument, JSON in the second argument, and finally the required callback in the third argument where you can log some feedback that will be displayed in the Terminal.

```
writeFile(resolve("./dist/package.json"), JSON.stringify(\
pkg, null, 2), () => {
  process.stdout.write("@in/ui package.json edited for di\
stribution \n");
});
```

When you are finished, `prepublish.js` should appear as follows.

```
import { readFile, writeFile } from "fs";
import { resolve } from "path";

readFile(resolve("./dist/package.json"), "utf8", (err, js\
on) => {
  const pkg = JSON.parse(json);
  pkg.main = "fesm2015/index.js";
  pkg.types = "types/index.d.ts";
  pkg.exports["."].es2015 = "./fesm2015/index.js";
```

```
  pkg.exports["."].default = "./fesm2015/index.js";
  pkg.exports["."].types = "./types/index.d.ts";
  delete pkg.scripts;
  writeFile(
    resolve("./dist/package.json"),
    JSON.stringify(pkg, null, 2),
    () => {
      process.stdout.write("@in/ui package.json edited fo\
r distribution \n");
    }
  );
});
```

This node script should run when lerna calls the `prepublishOnly` script in the `package.json` found at `packages/component/package.json`. Append `node prepublish.js` to the `prepublishOnly` script.

```
"prepublishOnly": "copyfiles package.json dist && node pr\
epublish.js"
```

The `prepublishOnly` step copies the `package.json` to the dist directory and runs the script in prepublish.js. This script reads the `package.json` in `dist`, sets the correct paths on the `main` and `types` fields, and deletes the `scripts` property because consumers of the package need not interpret these scripts used during the build process.

That's all there is to configuring the library build. In this section, you declared named exports to make the UI library tree-shakable. You configured TypeScript and Rollup to build a FESM which should optimize build times for downstream consumers of the package. TypeScript also output type definitions for developers to reference whenever importing anything from the `@in/ui` package. You reviewed how the package is built, inspecting the

`rollup.config.js`, `tsconfig.json`, and `package.json` and declared several properties on the `package.json` that enable bundlers to interpret the package. To resolve an issue with how lerna builds the `@in/ui` package and how consumers of the package should interpret the package, you implemented a script that edits the `package.json` for distribution only, while retaining the configuration lerna requires.

To run the `@in/ui` build with lerna, use the following command:

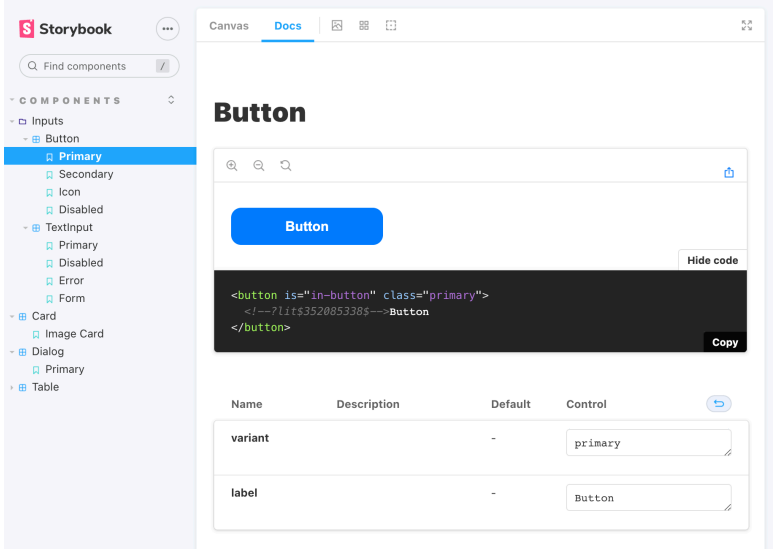
```
1  lerna exec --scope @in/ui -- yarn build
```

Providing Documentation with Custom Elements Manifest and Storybook

We've got a stable library package that could be published to npm so others can reuse the components. It should be simple, right? An engineer installs the `@in/ui` package and imports the component where applicable, but then is faced with a dilemma. How does any of this work? Savvy engineers may inspect the type definitions to get a sense of how a component works, but those lack any context for the HTML attributes which are a heavily used interface for custom elements. The engineer would then have to inspect the source, which is compiled to ES2015 and wrapped in a strange decorator pattern. To solve this dilemma, we can provide documentation for the end-users of the component library.

So far, you've used Storybook as a development environment. For each use case, you coded a Story and provided a template and arguments that generated Controls for the Storybook Canvas. Storybook also has built-in tooling to display documentation for each component in the Docs tab. The first Story for each component

is prioritized in the Docs tab. The user has the option to display the template in code beneath an example of the Story. When available, any HTML attributes or methods the component exposes are displayed in a table beneath the example.



By default, only the example is displayed for each Story in the Docs tab in Storybook. We need to provide additional documentation for each HTML attribute and method for Storybook to display contextual information about HTML attributes, methods, the tag name, or possibly description of the component. Storybook auto-generates the documentation with a tool called the Custom Elements Manifest. The `custom-elements-manifest` package analyzes each component `class` definition, parses HTML attributes, events, methods, the tag name, and description of the component and generates a JSON file which Storybook can use to configure the documentation for each Story. Custom Elements Manifest uses a CLI to run the script that generates the custom-elements.json file.

```
custom-elements-manifest analyze --dev --exclude "[./**/*\
.stories.js, dist]"
```

You'll eventually run the above command in the `packages/component` directory that produces a `custom-elements.json` file Storybook will interpret for parts of the documentation. To provide additional context around each HTML attribute or method, Storybook also interprets JSDoc comments.

JSDoc

JSDoc[73] is a standardized comment pattern for documenting JavaScript inline. Any comment written in JSDoc syntax above a particular code block documents the immediately proceeding code. For instance, a JSDoc comment above a `class` definition provides contextual information about the `class`, while a JSDoc comment above a method documents the method.

```
/**
 * Renders a dialog.
 * @tag {string} in-dialog
 * @param {string} style - css presentation
 */
@Component({
```

A document for getting started with JSDoc is available[74]. In short, the multi-line JSDoc comment begins with a description of the code block, followed by contextual tags. `function` arguments can be tagged with `@param`. HTML attributes can be denoted with `@attr`. After documenting each component with JSDoc, we'll have a format Storybook can interpret while generating the documentation for each component in the UI library. As an added bonus,

[73] https://jsdoc.app
[74] https://jsdoc.app/about-getting-started.html

TypeScript should compile type definitions with the inline JSDoc comments, so engineers can read the documentation right in their IDE.

Before generating the `custom-elements.json` file, document one component in the `@in/ui` package with JSDoc.

Starting with the `ButtonComponent` `class` in `packages/component/src/button/Buttor` document the `class` definition with JSDoc. After providing a description, use the `@tag` tag to define the tag name of the component. Use `@attr` tags to document the necessary HTML attributes that effect the display of `ButtonComponent`.

```
/**
 * Renders a button.
 * @tag {string} in-button
 * @attr {string} is - "in-button", Required
 * @attr {string} class - Variant, can either be "primary\
" "secondary", or "icon"
 */
@Component({
```

For Storybook to uptake these changes, you'll have to generate the `custom-elements.json` file which contains a JSON schema which Storybook uses to interpret each component for the documentation. We have a problem though, mostly caused by running Storybook in the context of a lerna monorepo. Storybook expects the `custom-elements.json` file to be in the root directory. If we ran the Custom Elements Manifest Analyzer in the `packages/component` directory, it would generate paths to each component implementation relative to that directory and not the root directory. The analyzer expects JavaScript, not TypeScript, so we have to build the `@in/ui` package first so the tool can parse the output in the `dist/fesm2015` directory. When that happens the `custom-elements.json` file will be written to `packages/component` with paths like in the following example:

```
{
  "kind": "javascript-module",
  "path": "dist/fesm2015/index.js",
  "exports": [
        {
                "kind": "js",
                "declaration": {
                        "name": "ButtonComponent",
                        "module": "./src/button"
                }
    }
  ]
```

Scripting the Build

You'll need to build the `@in/ui` package, run the analyze script, further process the `custom-elements.json` by replacing the paths so they are relative to the root directory, then write the `custom-elements.json` file to the root directory. That will allow Storybook to properly interpret the manifest and display documentation for each component.

The work begins in the `packages/component/package.json` file where the script is called. The `analyze` script already cleans the dist directory, builds the library and runs the analyzer.

```
"analyze": "yarn clean && yarn build && custom-elements-m\
anifest analyze --dev --exclude \"[./**/*.stories.js]\"",
```

Run the script using lerna.

```
lerna exec --scope @in/ui -- yarn analyze
```

Observe the `custom-element.json` file is outputted in the `packages/component directory`. If you open the file you'll see

paths relative to this directory. Storybook requires the same file to be placed in the root directory of the repository, so we have to update the paths programmatically everytime this script runs. To do that with node, make a new file in the `packages/component` directory named `manifest.js`.

```
touch packages/component/manifest.js
```

Open the `manifest.js` and import the `function` needed to read and write files and process paths.

```
import { readFile, writeFile } from "fs";
import { resolve } from "path";
```

Call `readFile`, passing in the three arguments required by the `function`: the path to the file, the format ("utf-8"), and the function callback where you'll process the contents of the file.

```
readFile(resolve("./custom-elements.json"), "utf8", (err,\
 content) => {});
```

You need to replace every path in the file wholesale and even though the file is JSON, we can treat the contents of the file as any text document. Use `String.prototype.replace` with `RegExp` statements that select each of the incorrect paths, and replace the string with the correct path.

```
readFile(resolve('./custom-elements.json'), 'utf8', (err,\
 content) => {
  content = content.replace(/\.\/src/g, './packages/compo\
nent/src');
  content = content.replace(
        /dist\/fesm2015/g,
        'packages/component/dist/fesm2015'
  );
```

Parse the content as JSON so it can be properly serialized for saving to a new file.

```
content = JSON.parse(content);
```

Call `writeFile`, passing in the three necessary arguments: the path the new `custom-elements.json` file should be written (the root directory of the monorepo), the JSON that could be written to the file, and the required callback where you can log some feedback for the engineer running the script.

```
writeFile(
  resolve("./../../custom-elements.json"),
  JSON.stringify(content, null, 2),
  () => {
    process.stdout.write("custom-elements.json saved to r\
oot \n");
  }
);
```

When you are finished coding the script, `manifest.js` should appear like below.

```
import { readFile, writeFile } from "fs";
import { resolve } from "path";

readFile(resolve("./custom-elements.json"), "utf8", (err,\
 content) => {
  content = content.replace(/\.\/src/g, "./packages/compo\
nent/src");
  content = content.replace(
    /dist\/fesm2015/g,
    "packages/component/dist/fesm2015"
  );
  content = JSON.parse(content);
  writeFile(
    resolve("./../../custom-elements.json"),
    JSON.stringify(content, null, 2),
    () => {
      process.stdout.write("custom-elements.json saved to\
 root \n");
    }
  );
});
```

Add the node script to the `analyze` script in the `package.json`, making sure to run the script after everything else.

```
"analyze": "yarn clean && yarn build && custom-elements-m\
anifest analyze --dev --exclude \"[./**/*.stories.js]\" &\
& node manifest",
```

Run the script with lerna.

```
lerna exec --scope @in/ui -- yarn analyze
```

Unfortunately Storybook cannot be running while the `custom-elements.json` file is being built. There is no mechanism

for hot-reloading changes to the file. If you have Storybook running, stop the server and restart Storybook.

```
yarn storybook
```

Adding Support for Custom Elements Manifest in Storybook

All that work and nothing happened! That's because we haven't configured Storybook to interpret the `custom-elements.json` file. By default, Storybook doesn't include this functionality and it's something you have to configure in `.storybook/preview.js`. Stop the Storybook server once again and open `.storybook/preview.js`. Add the following configuration to the file. Import `setCustomElementsManifest` from `@storybook/web-components` and the manifest itself from `../custom-elements.json`. Call `setCustomElementsManifest` with the JSON.

```
import { setCustomElementsManifest } from "@storybook/web\
-components";

import customElementsManifest from "../custom-elements.js\
on";

setCustomElementsManifest(customElementsManifest);
```

Restart Storybook and observe new documentation in the Button Story.

Writing Documentation

A description and two attributes are displayed with descriptions. Engineers may need more verbose documentation to get started

so add some more commentary surrounding the `ButtonComponent` in `packages/component/src/button/Button.ts`. A paragraph summarizing how the component is used should suffice, followed by verbose descriptions of each required attribute.

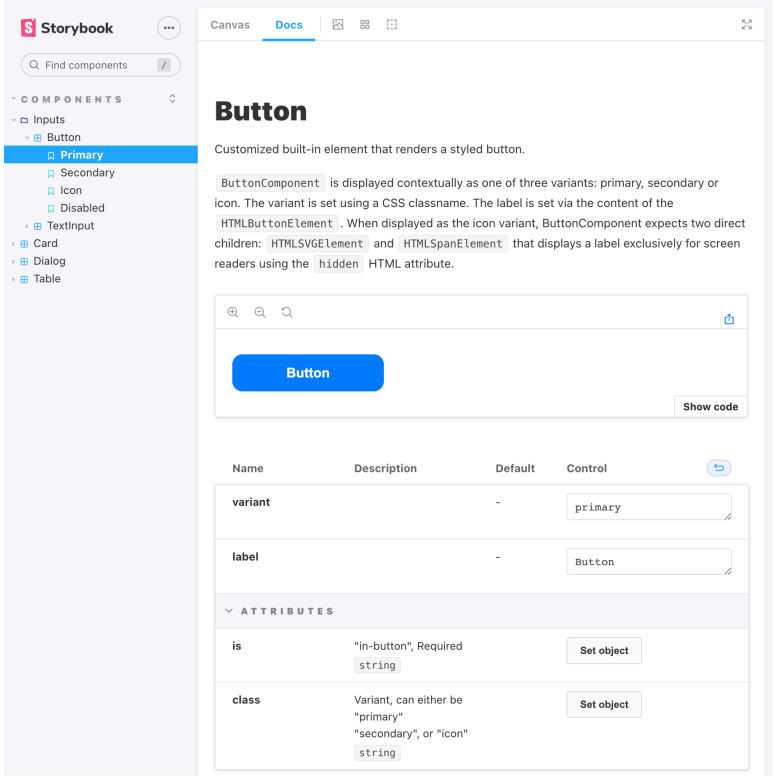
```
/**
 * Customized built-in element that renders a styled butt\
on.
 *
 * `ButtonComponent` is displayed contextually as one of \
three variants: primary, secondary or icon. The variant i\
s set using a CSS classname. The label is set via the con\
tent of the `HTMLButtonElement`. When displayed as the ic\
on variant, ButtonComponent expects two direct children: \
`HTMLSVGElement` and `HTMLSpanElement` that displays a la\
bel exclusively for screen readers using the `hidden` HTM\
L attribute.
 * @tag {string} in-button
 * @attr {string} is - "in-button", Required
 * @attr {string} class - Variant, can either be "primary\
" "secondary", or "icon"
 */
```

Stop the Storybook server and run the Custom Elements Manifest Analyzer again.

```
lerna exec --scope @in/ui -- yarn analyze
```

Start Storybook again and observe changes to the `ButtonComponent` documentation.

```
yarn storybook
```



Continue the same process with the other components until all of them are well documented. For each document, run the `analyze` script and restart Storybook to observe the changes.

Open `packages/component/src/card/Card.ts` to document `CardComponent`. Use the `@slot` tag to document each of the HTML template slots `CardComponent` requires for displaying content in the header, content, footer layout.

```
/**
 * Autonomous custom element that renders a card containe\
r.
 * `CardComponent` displays content in a header, content, \
 footer layout.
 * @tag {string} in-card
 * @slot {string} header - displays header elements
 * @slot {string} content - displays content in between h\
eader and footer
 * @slot {string} footer - displays footer elements
 */
```

After adding the JSDoc comment, running the script and restarting Storybook you may observe the documentation is still missing. That's because the Card Story isn't configured properly with the key value pair that links the Story to the component in the manifest. Open `packages/component/src/card/Card.stories.js` and add the required `component` property to the `Object` that specifies the Component Story Format.

```
export default {
  component: 'in-card',
  title: 'Components/Card',
```

Storybook should hot reload with the documentation. Open `packages/component/src/dialog/Dialog.ts` next and write some documentation about the `DialogComponent`.

```
/**
 * Controller for rendering a dialog from HTML template.
 *
 * `DialogComponent` requires an instance of `DialogStack\
` to be set on `window.__dialogStack` prior to instantiat\
ion. After specifying a `template`, `target`, and `varian\
t` with HTML attributes, `DialogComponent` will display e\
ither a modal or tooltip that renders a HTML template.
 * @tag {string} in-dialog
 * @attr {string} template - Required, Selector that targ\
ets the HTML template, for use with `querySelector`.
 * @attr {string} target - Selector that targets the HTML\
 element that triggers the dialog on click, for use with \
`querySelector`, Required
 * @attr {string} variant - Specifies whether to display \
a "modal" or "tooltip", Required
 */
```

Continue with the necessary documentation for `TextInputComponent` in `packages/component/src/input/TextInput.ts`.

```
/**
 * Renders an input element.
 * @tag {string} in-textinput
 * @attr {string} name - Name of the form control. Submit\
ted with the form as part of a name/value pair
 * @attr {string} type - Type of form control
 * @attr {boolean} required -  A value is required or mus\
t be check for the form to be submittable
 * @attr {string} minlength - Minimum length (number of c\
haracters) of value
 * @attr {string} maxlength - Maximum length (number of c\
haracters) of value
 * @attr {string} pattern - Pattern the value must match \
to be valid
```

```
 * @attr {string} list - Value of the id attribute of the\
 <datalist> of autocomplete options
 * @attr {string} placeholder - Text that appears in the \
form control when it has no value set
 * @attr {boolean} readonly - The value is not editable
 * @attr {string} size - Size of the control
 * @attr {string} spellcheck - Defines whether the elemen\
t may be checked for spelling errors.
 */
```

Follow up with documentation for `TableCardComponent` in `packages/component/src/table/TableCard.ts`.

```
/**
 * Renders a `TableComponent` in a stylized card
 * that displays buttons used for CRUD in the table.
 * @tag {string} in-tablecard
 * @attr {string} channel - the name of the channel used \
to communicate via `BroadcastChannel`, Required
 */
```

Finally, add documentation for `TableComponent` in `packages/component/src/table/Table.ts`.

```
/**
 * Renders a stylized `HTMLTableElement` that can display\
 read-only or editable.
 * `TableComponent` is a customized built-in element that\
 renders columns and rows through events dispatched on `B\
roadcastChannel`.
 * @attr {string} channel - the name of the channel used \
to communicate via `BroadcastChannel`, Required
 * @attr {string} is - "in-table", Required
 */
```

Documentation is necessary to secure the successful integration of any custom element UI library. With clear, concise documentation, engineers are able to quickly understand how custom elements are supposed to be used. Storybook reinforces the documentation with code samples mapped to each Story. All this documentation would go to waste if it only ran locally. In the next section, you'll establish configuration that allows Storybook to run remotely in Github Pages using another Github tool: Github Actions.

Set up Continuous Integration with Github Actions

Right now, packages in the monorepo can only be built locally on a developer's computer. There's no way to build or test the UI library remotely, which could cause problems not only for other members of the development team, but those engineers who rely on the UI library for development. If an engineer checked in code to the main branch of the repo that couldn't be built, it may cause unwanted churn for others on the team. Anyone who wants to reference the documentation would have to clone the repository and build it locally. Continuous integration (CI) is a software development practice that automates the process of running builds, validating code against tests, and deploying resources like packages or servers. CI is beneficial for UI libraries because it provides transparency and insight into the development cycle of the library, while maintaining quality and consistency.

There are many tools available for Continuous Integration, but we're going to focus on getting CI up and running for the UI library using Github Actions. You already check in code to a repository hosted on Github for the purposes of completing this book, so it makes sense to use the tool that is native to the Github platform. Github Actions automates software development workflows whenever an engineer pushes code to a branch in Github. The process can

be visualized in pull requests. When an engineer pushes up new commits to a branch associated with a pull request, Github Actions start whatever workflow is run `on: [push]`.

Github Actions are configured using YAML[75], "a human-friendly data serialization language for all programming languages" according to the yaml.org. A single Github Actions workflow has a name, is performed on an event, and can run several jobs. Jobs have steps that run in parallel or sequentially. A complete guide for Github Actions is available here[76].

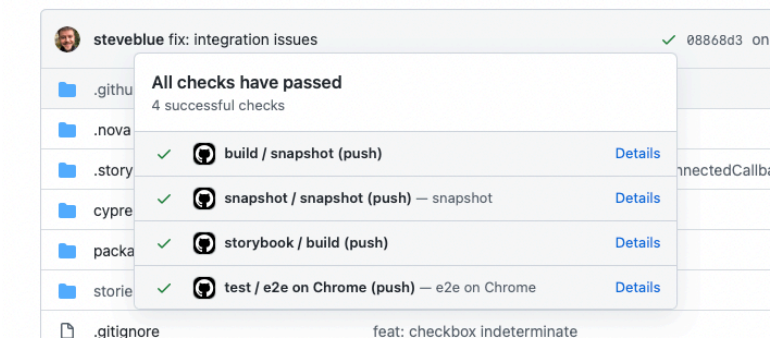
In this section you'll write YAML to configure four Github Actions workflows:

- Build validates that all packages in the monorepo can be built without error (on push)
- Test runs integration tests (on push)
- Snapshot runs snapshot tests (on push)
- Storybook builds and deploys Storybook to Github Pages (on push in main branch only)

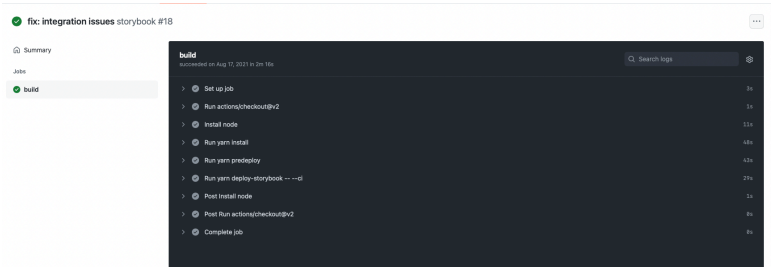
If you're not familiar with YAML, don't worry. A walkthrough for setting up each Github Action is provided along with step-by-step instructions. To validate the Github Actions workflows are working correctly, you'll need to make changes to each `.yaml` file, commit the changes and push up the branch to your fork of the `fullstack-web-components` repository. Open a pull request to merge your branch into the main branch of the forked repo. You should be able to observe Github Actions workflows run on the last commit message in the PR.

Click on the icon in the last commit to view the current status of the workflow. When a pull request is merged to the main branch, an icon appears there as well to display the status of Github Actions workflows associated with pushing to the main branch.

[75] https://yaml.org
[76] https://docs.github.com/en/actions/learn-github-actions



When clicking on details of any Github Actions workflow, you should be able to observe the output of the jobs in that workflow.



Build

The first Github Actions workflow you'll configure is the build. To start on this workflow, make a new directory named `.github` in the root directory of the monorepo, making sure to preface the directory name with `.` which infers the directory is hidden.

```
mkdir .github
```

Inside of the `.github` directory, make another directory named `workflows`. All the Github Actions workflows will be saved in this directory.

```
mkdir .github/workflows
```

Make a new YAML file named `build.yml` in `.github/workflows`.

```
touch .github/workflows/build.yml
```

Open `build.yml` in your favorite IDE to begin configuring the build Github Actions workflow. Start by specifying the name of the workflow: `build`. YAML is written in succinct key/value pairs separated by a colon. On the second line of the `build.yml` file, declare the events this workflow should run: `on: [push]`, meaning whenever new commits are pushed to any remote branch.

```
name: build
on: [push]
```

Next, specify the `jobs` that should run on this workflow. There is only one job: `build` which is declared as a key on `jobs`. The `build` needs to run on a remote server. The latest Ubuntu linux distribution should work. Declare the `steps` of the job. Each step should use the `actions/checkout@v2`, which automates the process of checking out a commit and running the job. Ubuntu doesn't come with NodeJS installed, so you need to also specify to install `node` and `yarn`. After the installation is completed, it's safe to `run` two scripts in the monorepo: `yarn install` which installs all project dependencies and `yarn build` which calls on lerna to build every package in the monorepo.

```
jobs:
        build:
                runs-on: ubuntu-latest
                steps:
                  - uses: actions/checkout@v2
                  - name: Install node
                        uses: actions/setup-node@v2
                        with:
                          node-version: "16"
                          cache: "yarn"
                  - run: yarn install
                  - run: yarn build
```

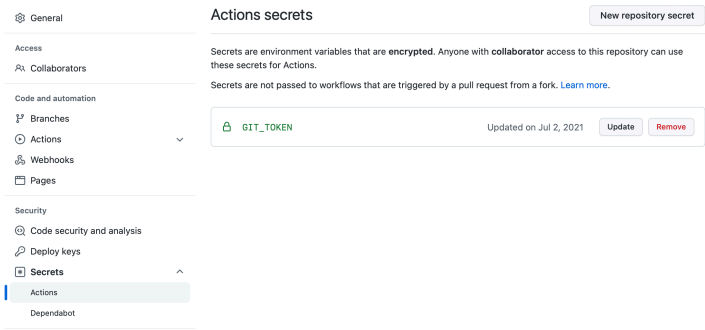
When you are done, the workflow should appear as follows:

```
name: build
on: [push]
jobs:
        build:
                runs-on: ubuntu-latest
                steps:
                  - uses: actions/checkout@v2
                  - name: Install node
                        uses: actions/setup-node@v2
                        with:
                          node-version: "16"
                          cache: "yarn"
                  - run: yarn install
                  - run: yarn build
```

Commit your changes on the `chapter-7` branch and push them to the remote fork of the git repository. Make a new pull request for the main branch of the forked repository and observe that the Github Action is running on the last commit of the PR.

Integration Tests

Setting up Cypress integration tests in Github Actions is a little more involved, but involves many of the same techniques found in the build workflow. Cypress provides a tool for running tests in Github Actions named `cypress-io/github-action@v2` and like `actions/checkout@v2` and `actions/setup-node@v2` in the previous workflow, the job will have to use `cypress-io/github-action@v2`. `cypress-io/github-action@v2` relies on a Github Actions Token that should be installed in Github prior to running the workflow. Navigate to the **Settings** view for your fork of the `fullstack-web-components` repository. Find the **Secrets** menu under **Security** in the sidebar and click **Actions**. Click on **New repository secret**. Give the secret a name. I used the name "GITHUB_TOKEN", followed by a value, which can literally be anything. Click **Add Secret** when you are finished.



On your local branch, make a new file to configure the test Github Actions workflow.

```
1  touch .github/workflows/test.yml
```

In `test.yml`, give the workflow a name and specify when it should run, `on: [push]`. In `jobs`, give the job a name

`cypress-run`, that runs on `ubuntu-latest`, like the previous workflow. Give the job a name that will be displayed in the Github Actions UI: `e2e on Chrome`. In the `steps`, provide `actions/checkout@v2` and `cypress-io/github-action@v2`. `cypress-io/github-action@v2` requires a significant amount of configuration including how to install dependencies `install-command: yarn install`, which browser to run the e2e tests `browser: chrome`, the name of the repository: `repository: [github username]/fullstack-web-components` (replace `[github username]` with your Github username), a branch to reference for failing tests `ref: refs/heads/master`, the token you specified in Github `token: ${{ secrets.GITHUB_TOKEN }}`, a setting that disables the persistence of credentials `persist-credentials: false`, the command that builds the project `build: yarn build`, the command that starts the Storybook server `start: yarn start` and finally, the address where Storybook runs Cypress, should listen and wait for before running the tests `wait-on: http://localhost:6006`.

```
name: test
on: [push]
jobs:
  cypress-run:
        runs-on: ubuntu-latest
        name: e2e on Chrome
        steps:
          - uses: actions/checkout@v2
          - uses: cypress-io/github-action@v2
                with:
                  install-command: yarn install
                  browser: chrome
                  repository: [github username]/fullstack-web-components
                  ref: refs/heads/master
                  token: ${{ secrets.GITHUB_TOKEN }}
                  persist-credentials: false
```

```
17                  build: yarn build
18                  start: yarn start
19                  wait-on: http://localhost:6006
```

After you've configured `test.yml` like the above example, being sure to replace `[github username]` with your Github username, commit your changes and push them to the branch currently associated with a pull request. Observe the workflow runs in Github Actions.

Snapshot Tests

Make another workflow file named `snapshot.yml` in the `.github/workflows` directory.

```
1  touch .github/workflows/snapshot.yml
```

Name the workflow `snapshot` and specify that it happens on push of any branch.

```
1  name: snapshot
2  on: [push]
```

Storyshots requires a similar workflow to the Cypress integration tests, although it's much harder to configure because there isn't a package like `cypress-io/github-action@v2` for Storyshots that makes configuration easier in Github Actions. This means you have to manage the installation of node and yarn, the command that runs in the job much more granualarly. `actions/upload-artifact@v2` does exist though, which automates the upload of any artifacts from the snapshot tests. You can use `actions/upload-artifact@v2` to temporarily upload any snapshot that failed to observe what may have gone wrong. Any snapshot with a "diff" is stored at `packages/component/src/**/__snapshots__/__diff_output__/*`.

Another important setting in this job is `continue-on-error: true` which forces the workflow to complete despite any snapshot failing, so you could observe all the failing snapshots instead of just the first that failed.

```
name: snapshot
on: [push]
jobs:
  snapshot:
        runs-on: ubuntu-latest
        steps:
          - uses: actions/checkout@v2
          - name: Install node
                uses: actions/setup-node@v2
                with:
                  node-version: "16"
                  cache: "yarn"
          - run: yarn install
          - run: yarn test:snapshot:ci
                continue-on-error: true
          - name: Upload artifacts
                uses: actions/upload-artifact@v2
                with:
                  name: snapshot-diff
                  path: packages/component/src/**/__snapshots__/__diff_\
output__/*
```

When you have completed in `snapshot.yml`, commit and push your changes to the remote branch. Observe the snapshot tests running in Github Actions.

Deploy Storybook

To make a new workflow for deploying Storybook to Github Pages, make a new file at `.github/workflows/storybook.yml`.

```
touch .github/workflows/snapshot.yml
```

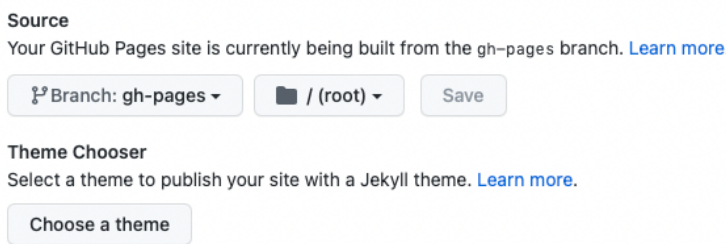
The workflow for deploying Storybook is very similar to the build workflow, with some exceptions. We want to only deploy Storybook to Github Pages when a commit is pushed to the main branch, not just any branch like previous workflows. You can specify the specific `branches` you wish to run the workflow: `main`. Run the jobs on `ubuntu-latest`, being sure to install `node` and `yarn`. Install all project dependencies with `yarn install`. After that, two things need to happen: Storybook should be built and then deployed. A script in the main `package.json` of the repository handles `predeploy` tasks, essentially building Storybook. A second `script` named `deploy-storybook` uses a CLI from the `@storybook/storybook-deployer` package to deploy Storybook to Github Pages. That script needs to be run with the `--ci` argument and uses the token you specified before for the repository, only formatted slightly differently.

```
name: storybook
on:
  push:
        branches:
          - main
jobs:
  build:
        runs-on: ubuntu-latest
        steps:
          - uses: actions/checkout@v2
          - name: Install node
                uses: actions/setup-node@v2
                with:
                  node-version: "16"
                  cache: "yarn"
          - run: yarn install
          - run: yarn predeploy
```

```
18            - run: yarn deploy-storybook -- --ci
19                  env:
20                    GH_TOKEN: ${{ github.actor }}:${{ secrets.GITHUB_TOKE\
21  N }}
```

Commit your changes and push them to the remote branch. Observe the rest of the workflows run except this one in the pull request. Once Github Actions validates the build, integration and snapshot tests, in your browser, merge the branch to the main branch of your remote fork. The Storybook workflow should run on the main branch. You can observe its progress on the last commit that was merged on the main page of the repository. Once the workflow has completed, the `deploy-storybook` script should have made a new branch named `gh-pages` where the current deployment of the Storybook is stored. Github Pages references the `gh-pages` branch for changes and updates the Storybook portal whenever the `gh-pages` branch is updated.

To enable Github Pages you still need to navigate to the Settings for the Github repository in your browser. Open **Settings** and click on **Pages** in the sidebar.



Specify the `gh-pages` branch as the source and click **Save**.

It may take another roundtrip merging to the main branch to finalize the process now that Github Pages is enabled via the Github UI in your browser. When Storybook is properly deployed it will be

visible at [github username].github.io/fullstack-web-components.

After writing these four workflows for Github Actions, you are now able to validate your changes in a continuous integration pipeline. One workflow we omitted was deployment of the `@in/ui` and `@in/common` packages to the global npm registry. Given what you know now about Github Actions, that shouldn't be too hard to figure out.

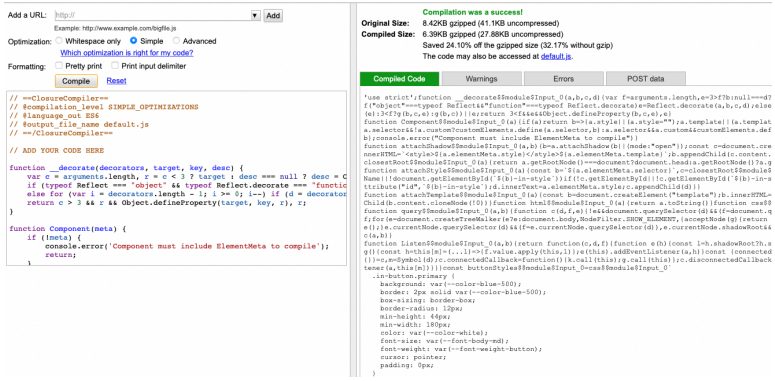
Summary

In this chapter, you took a break from coding components and learned about maintenance of a UI library built with custom elements. You exported components with named 2015 exports so the UI library can be treeshaken by build tools like Webpack and Rollup. You optimized the distributed code by bundling it into one file. By distributing a FESM, build tools only have to interpret the exports from one file as opposed to many. This should speed up build times because build tools do not have to statically analyze the AST sourced from many different files. Next, you discovered how to write documentation for each component using JSDoc and provided rich documentation with Storybook and Custom Elements Manifest. Finally, you set up a continuous integration workflow using Github Actions that builds and validates the codebase, while deploying Storybook to Github Pages using automation.

All this work will come in handy for the final chapters of this book, where you will develop a single page application using the custom elements exported from the `@in/ui` package. You've come a long way since Chapter One. After learning about the various types of custom elements, you built several complex user interfaces using the complete set of specifications known as Web Components: custom elements, HTML templates, and ES Modules. In Part Two, you learned how components can communicate with HTML attributes, `CustomEvent` and `BroadcastChannel` while building the table and

dialogs. You coded a micro library that reduced the repetitive tasks associated with custom element development and gave engineers easy-to-use interfaces for declaring tag name, template, styling, and events.

All this hard work wasn't for nought. Through a combination of using a custom micro-library and sticking only with browser specifications, the complete UI library you've built so far optimizes to ~6.4kB gzipped when optimized with Closure Compiler in `SIMPLE_OPTIMIZATIONS` mode and could be optimized further with additional tooling that minifies any template strings. You can test the size of the bundle yourself by pasting the contents of it into the portal at https://closure-compiler.appspot.com along with specifying the `language_out` should be `ES6`.



You've demonstrated the potential of Web Components by optimizing the way they are distributed and documented. Onward to Part Three where you will develop an entire application using only custom elements and other browser specifications.

Part Three

Part Three

Application

In Part Three of this book, you'll explore how Web Components can make building applications easier and performant. By reusing the components distributed in the `@in/ui` package, you'll code feature-rich views for the "In" application. The web app allows the user to login and interact with a contact list. You'll start by coding the login view. After developing a splash page and login user interface, you'll switch gears and code a router that can safely destroy the view and instantiate another: the "In" contact list. Throughout Chapters 8-10, you'll not just learn how to reuse components from the UI library but also how to make an application performant by server-side rendering custom elements where applicable using Declarative Shadow DOM, the final specification in Web Components you have yet to leverage.

What you will learn

While developing the In application with Web Components, you'll explore the following concepts:

- How to build features in a single page application using Web Components
- How to provide a router that can dynamically add and remove views built with custom elements

- What is Declarative Shadow DOM?
- How to server-side render custom elements with Node
- How to ensure a high percentage lighthouse performance score

Developing in Part Three

In previous chapters, you used Storybook as the primary development environment. In Part Three, you'll code the front end of the application in the `packages/client` directory and the backend of the application in the `packages/server` directory. A script can be executed by running `yarn start:app` in the root directory of the repository. This script bootstraps both client and backend servers and listens for changes while you are developing, and reloads the browser window on every change. After a successful development build, the server runs at http://localhost:4443. A webpack[77] server drives client-side development, while ts-node[78] and nodemon[79] handle the Node.js server. Client code is written in TypeScript, while server code is written in JavaScript. JavaScript was chosen for the server to maintain stability for the lessons in this book.

If you wish to test the production build during development, you may run `yarn build` in both client and server packages using lerna[80], i.e. `lerna exec --scope @in/client -- yarn build && lerna exec --scope @in/server -- yarn build`. Once built, the production server is invoked with `lerna exec --scope @in/server -- yarn start:prod`. The production server runs at http://localhost:4443.

In previous chapters, you ran `yarn storybook`, which may still be useful for the remaining chapters of this book if you want to reference the documentation of each component via Storybook. To

[77] https://webpack.github.io
[78] https://typestrong.org/ts-node/
[79] https://nodemon.io
[80] https://lerna.js.org

run Storybook simultaneously, open a second tab (or window) in your Terminal and run `yarn storybook` while `yarn start:app` is running in the first tab. In Part Three, you'll mainly run the script `yarn start:app` for development.

Application Development with Custom Elements

Chapter Eight

Application Development with Custom Elements

Application development with custom elements is not unlike application development with any other library or framework. JavaScript libraries prescribe various ways of working. React, for example, prefers a functional implementation, while Vue applications export an Object-based configuration in each file. The absence of such framework-level abstractions can be intimidating for anyone attempting to develop an application purely with Web Components. How you architect an application with vanilla JavaScript and browser specifications is up to you, but the rewards are great. By minimizing library dependencies, you can ensure the application bundle(s) are small in file size. Less JavaScript for the browser to interpret as the page loads can translate to a more performant solution. Users would have less time to wait to interact with content.

There is nothing stopping you from using existing frameworks as inspiration when developing with custom elements. Anyone who has coded with Angular may have noticed the similarity between Angular's `class` decorators and the design of the `@Component` decorator exported from `@in/ui`. You could come up with low-level functions that enable application development similar to any UI

framework you're comfortable with. In this chapter, you'll implement patterns for application development using custom elements that are only one opinion of how an application could be developed. The possibilities are seemingly endless! Specifically, you'll develop the landing page of the application using custom elements and class-based services that communicate with backend APIs over http requests.

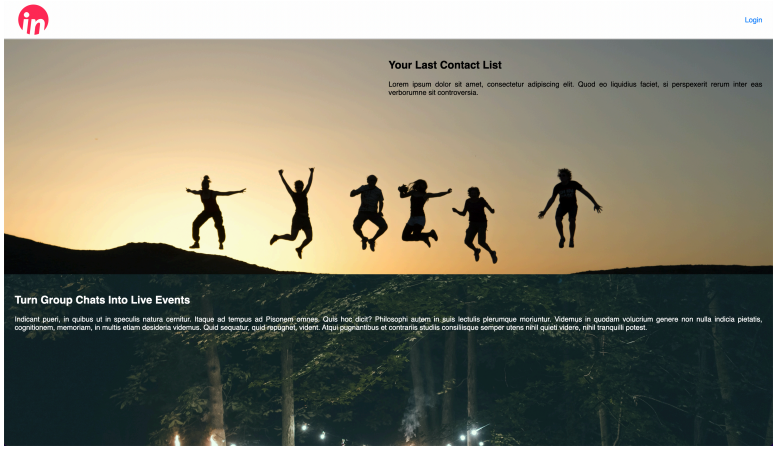
During the development of complex user interfaces like `TableCardComponent`, you implemented a separation of concerns between the various user interface elements: table card, table, row, and cell. We still want to maintain a separation of concerns during application development to ensure the product can scale. For this reason, we'll abstract any HTTP requests made to server-side REST APIs into their own service-level pattern.

```
export class CookieService {
  private path = "/api/cookies";
  constructor() {}
  getPermission(): Promise<CookiePermission> {
    ...
  }
}
```

Each service is a `class` that exposes methods for fetching data or submitting data to the API endpoints the service is concerned with. In the above example, `CookieService` exposes a method named `getPermission` that returns a `Promise` (the result of the http request). `CookieService` sets and returns user-specified permission for using cookies, while `SessionService` should report if the user has a current session, for example.

In addition to this service-level abstraction, you'll abstract reusable application-level UI components to discrete implementations. `AppHeader` should be displayed in every view of the application, so it makes sense to abstract its implementation so it can be reused.

`CookieFooter` is another UI component that should display a footer with messaging for the end-user to accept or deny cookies for the site. `Background` is a UI component that lazyloads a background image for each section of the landing page.



`AppHeader`, `CookieFooter` and `Background` will be used in the template for the landing page view. We'll maintain application views as separate UI components. In this chapter, the `MainView` is another autonomous custom element mainly concerned with implementing a template for the entire landing page. In the next chapter, `MainView` will be attached to a `Route` when you build out a client-side router compatible with custom elements. By providing a separation of concerns sooner than later, we can anticipate the challenges ahead in a multi-view application. For now, we're only concerned with implementing the landing page.

What you will build

- Landing page for "In" application using custom elements
- `AppHeader` which is the global header for the application

- `CookieFooter` which displays a banner the user can accept or deny cookies
- `Background` which lazyloads a background image in a container element
- `MainView` which displays the entire view for the landing page
- `SessionService` which reports if the user has a current session
- `CookieService` which sets and returns user-specified permission for using cookies

What you will learn

- How to make a header, content, and footer template with custom elements
- How to code a reusable global header with custom elements
- How to lazyload images with customized built-ins
- How to code a cookie banner with custom elements
- How to reuse custom elements from a UI library during application development
- How to create a separation of concerns between components and view
- How to make http requests from the context of a custom element using `fetch`

Acceptance criteria

- `MainView` displays all content according to the mockups in Figma
- `AppHeader` displays a logo that navigates the user to `MainView` (/)
- `AppHeader` displays a link that navigates the user to `LoginView` when the user doesn't have an active session

- `CookieFooter` displays a message and two buttons, a primary button that displays "Accept" and secondary button that displays "Deny"
- When the user clicks "Accept", `CookieBanner` is not visible until the backend responds with `permission: false`
- When the user clicks "Deny", `CookieBanner` remains visible
- `Background` lazyloads background images in both sections
- `SessionService` has a method UI components can use to check if the user has an active session
- `CookieService` has methods to check the status of the user preference and set the user preference for cookies

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-8 from where you left off at the end of chapter 7.

```
git checkout -b chapter-8
```

Landing Page

To begin development of the landing page, we should implement the view. The view is just another custom element, so we'll import methods from `@in/common` and establish a template and styling for the entire view. Since any UI components displayed on the landing page will be in the template of `MainView`, we can use this component as a development environment of sorts for the

remainder of this chapter. One difference between the custom elements in the application and user library developed in previous chapters is we're going to declare the template and styling for each component outside of the `class` and then set the `style` and `template` properties on the `@Component` decorator. This will help us refactor the components for server-side rendering in Chapter Twelve.

All development in this chapter happens in `packages/client/src`. A directory named `view` has already been scaffolded in here, so create a file named `MainView.ts` in this directory, then open the file with your favorite IDE. Let's make another directory named `main` and put `MainView.ts` in there.

```
mkdir packages/client/src/view/main
touch packages/client/src/view/main/MainView.ts
```

In `MainView.ts`, import the necessary `function` and decorators from `@in/common` needed for an autonomous custom element.

```
import { attachShadow, html, css, Component } from "@in/c\
ommon";
```

Declare two `const` for the CSS styling and template for the component named `styles` and `shadowTemplate` respectively. You can set these to template strings, calling the appropriate `css` or `html` function exported from `@in/common`.

```
const styles = css``;
const shadowTemplate = html``;
```

Using the `@Component` decorator, declare a new custom element with the selector `main-view` and set the `style` and `template` properties to the two `const` you declared prior. Export a new `class` named `MainView` that extends `HTMLElement`, like any other autonomous custom element. Opt to use Shadow DOM for this component in the `constructor` by calling `attachShadow`.

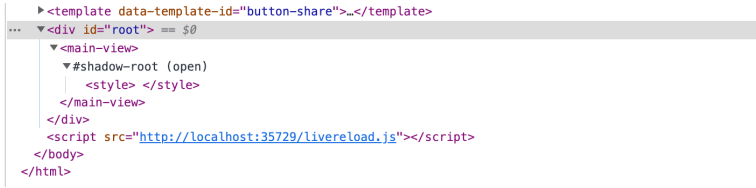
```
@Component({
  selector: "main-view",
  style: styles,
  template: shadowTemplate,
})
export class MainView extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

If you haven't started the server yet, do so now by running `yarn start:app` and open http://localhost:4443 in your preferred development browser. Nothing should appear yet. That's because we haven't injected an instance of `MainView` into DOM. Open `packages/client/src/index.ts`. Using `document.createElement`, create a new instance of `MainView`. If you inspect the DOM, you'll notice an empty `HTMLDivElement` with the `id` attribute set to `root`. Select that element and append the instance of `MainView` to it. Export `MainView` from this file. `index.ts` is the entry point of the entire client-side application, so anything exported here gets bundled with the application.

```
const mainView = document.createElement("main-view");
document.querySelector("#root").appendChild(mainView);

export { MainView } from "./view/main/MainView";
```

There's not much happening yet, but at the very least, the component is now in DOM.



Add a header, content, and footer template to `shadowTemplate`, being sure to set an attribute to select the content. Here we use semantic HTML elements and a `HTMLDivElement` with `id` attribute set to `content-root`.

```
const shadowTemplate = html`
  <header></header>
  <div id="content-root"></div>
  <footer></footer>
`;
```

If you want, add some temporary content to each element to visualize the style updates.

```
const shadowTemplate = html`
  <header>Header</header>
  <div id="content-root">Content</div>
  <footer>Footer</footer>
`;
```

Add some basic styling that establishes style behaviors for the containing element. We want the header to display at the top, while the footer displays at the bottom, and the content scrolls in between. This layout can be achieved with flex box. After selecting `:host`, set the `display` to `flex`, `flex-direction` to column so the layout flows vertically, `justify-content` to `space-between` so the elements flow where we want them, and set the `height` to always be 100% of the view: `100vh`. To allow the content area to scroll, select `#content-root` and set `overflow-y` to scroll.

```
const styles = css`
  :host {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    height: 100vh;
  }
  #content-root {
    overflow-y: scroll;
  }
`;
```

You'll circle back to update the `MainView class` throughout the rest of this chapter, but for now, you have a foundation to develop the `AppHeader`, `CookieFooter` and `Background` components needed to display the content of the landing page. In the next section, you'll develop the `AppHeader` while learning how to abstract HTTP requests to a service.

App Header

In this section, you'll build out the `AppHeader`, a UI component that displays at the top of every view. `AppHeader` displays the corporate logo and link that navigates to the `LoginView` you have yet to develop. This link on the right side of `AppHeader` only should appear if the user isn't authenticated with the server. The backend provides a REST API endpoint at http://localhost:4443/api/session that responds if the user currently has a session with the backend after the user has authenticated. When the user clicks on the logo, the browser should navigate to the `MainView`.

You could make the HTTP request to the `api/session` endpoint from the context of `AppHeader`, but to ensure the application can scale and other components can make similar requests, you'll code

a `SessionService` that contains the logic for making the HTTP request with `fetch` and serializes the response for consumption in UI components. The serialized response should be type defined for reuse as well.

First, you'll scaffold `AppHeader`, declaring a template and styling for the component and then develop `SessionService` in parallel. Make a new directory for application components at `packages/client/src/component`.

```
mkdir packages/client/src/component
```

Put a directory named header in `packages/client/src/component`.

```
mkdir packages/client/src/component/header
```

Make a new file named `AppHeader.ts` in the header directory.

```
touch packages/client/src/component/header/AppHeader.ts
```

Import the necessary `function` and decorators from `@in/common` to develop an autonomous custom element.

```
import { attachShadow, html, css, Component } from "@in/c\
ommon";
```

Declare two `const` that implement template strings where you'll declare styling and template for `AppHeader`.

```
const styles = css``;
const shadowTemplate = html``;
```

Using the `@Component` decorator, declare a new custom element with the selector `app-header` and set the `style` and `template` properties to the two `const` you declared earlier. Export a new `class` named `AppHeader` that extends `HTMLElement` Use Shadow DOM for this component in the `constructor` by calling `attachShadow`.

```
@Component({
  selector: "app-header",
  style: styles,
  template: shadowTemplate,
})
export class AppHeader extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

Make a new file named `index.ts` in `packages/client/src/component` that serves as an index for all application components.

```
touch packages/client/src/component/index.ts
```

Export `AppHeader` from packages/client/src/component/index.ts

```
export { AppHeader } from "./header/AppHeader";
```

Export `AppHeader` from packages/client/src/index.ts.

```
export { AppHeader } from "./component";
```

While you're working in `packages/client/src/index.ts`, export the necessary component from `@in/ui` you'll be working with in `AppHeader`. The icon on the left side of the global header should be an instance of `ButtonComponent`. when you are done, `index.ts` should appear like below.

```
const mainView = document.createElement("main-view");
document.querySelector("#root").appendChild(mainView);

export { ButtonComponent } from "@in/ui";
export { AppHeader } from "./component";
export { MainView } from "./view/main/MainView";
```

Template

Now `AppHeader` is exported from the application entry point, you should be able to develop with it, but first update the `MainView` template, replacing the `HTMLHeaderElement` with the `AppHeader` custom element.

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root"></div>
  <footer></footer>
`;
```

In `packages/client/src/component/header/AppHeader.ts`, add an icon button and link to the template.

```
const shadowTemplate = html`
  <button is="in-button" class="in-button icon logo"></bu\
tton>
  <a href="/login" class="login-link">Login</a>
`;
```

Add the SVG icon to the template.

```
const shadowTemplate = html`
  <button is="in-button" class="in-button icon logo">
    <svg
      width="1024"
      height="1024"
      viewBox="0 0 1024 1024"
      fill="none"
      xmlns="http://www.w3.org/2000/svg"
    >
      <circle cx="512" cy="512" r="512" fill="#FF2D55" />
      <path
        d="M345.544 493.056L282.056 1024H125.384L188.872 \
493.056H345.544ZM375.24 347.648C375.24 360.277 372.509 37\
2.224 367.048 383.488C361.587 394.411 354.419 404.139 345\
.544 412.672C336.669 420.864 326.429 427.52 314.824 432.6\
4C303.56 437.419 291.955 439.808 280.008 439.808C268.061 \
439.808 256.627 437.419 245.704 432.64C234.781 427.52 225\
.053 420.864 216.52 412.672C208.328 404.139 201.672 394.4\
11 196.552 383.488C191.432 372.224 188.872 360.277 188.87\
2 347.648C188.872 335.019 191.432 323.072 196.552 311.808\
C201.672 300.544 208.499 290.645 217.032 282.112C225.907 \
273.579 235.805 266.923 246.728 262.144C257.651 257.024 2\
69.085 254.464 281.032 254.464C293.32 254.464 305.096 256\
.853 316.36 261.632C327.624 266.411 337.523 273.067 346.0\
56 281.6C354.931 290.133 361.928 300.032 367.048 311.296C\
372.509 322.56 375.24 334.677 375.24 347.648ZM583.92 584.\
704C609.52 550.571 637.509 525.483 667.888 509.44C698.267\
 493.056 729.669 484.864 762.096 484.864C783.941 484.864 \
803.909 489.301 822 498.176C840.432 507.051 855.621 520.0\
21 867.568 537.088C879.856 554.155 888.56 575.317 893.68 \
600.576C898.8 625.493 899.312 654.165 895.216 686.592L855\
.28 1024H696.56L736.496 686.592C738.203 672.256 738.715 6\
60.139 738.032 650.24C737.691 640.341 735.984 632.491 732\
.912 626.688C729.84 620.885 725.573 616.789 720.112 614.4\
C714.651 611.669 708.165 610.304 700.656 610.304C689.051 \
```

```
610.304 676.933 613.888 664.304 621.056C651.675 628.224 6\
39.557 638.464 627.952 651.776C616.688 664.747 606.277 68\
0.619 596.72 699.392C587.163 717.824 579.653 738.304 574.\
192 760.832L545.008 1024H386.288L449.776 493.056H531.696C\
548.08 493.056 560.88 496.981 570.096 504.832C579.312 512\
.341 583.92 524.8 583.92 542.208V584.704Z"
        fill="white"
      />
      <mask
        id="mask0_102_150"
        style="mask-type:alpha"
        maskUnits="userSpaceOnUse"
        x="0"
        y="0"
        width="1024"
        height="1024"
      >
        <circle cx="512" cy="512" r="512" fill="#FF2D55" \
/>
      </mask>
      <g mask="url(#mask0_102_150)"></g>
    </svg>
  </button>
  <a href="/login" class="login-link">Login</a>
`;
```

Styling

Define some basic styling for `AppHeader`. Use flexbox again to justify and align the icon button and link. Give the element a specific height and apply the padding and border specified in Figma.

```
const styles = css`
  :host {
    display: flex;
    justify-content: space-between;
    align-items: center;
    height: 100px;
    padding-left: var(--margin-lg);
    padding-right: var(--margin-lg);
    border-bottom: var(--border-default);
  }
`;
```

Correct some of the styling for the icon so it appears per the mockup.

```
:host .in-button.icon.logo {
  display: block;
  border: none;
  width: 84px;
  height: 84px;
}
```

Add styling for the links because global styling cannot penetrate the shadow boundary.

```
a:link,
a:visited {
  font-family: var(--font-default);
  font-weight: var(--font-weight-default);
  font-size: var(--font-body-md);
  text-decoration: none;
  color: var(--color-blue-500);
}
```

After confirming the link is styled, add the `hidden` attribute on the element. This will hide the element by default, which is the desired outcome. The link should only appear if the user isn't authenticated.

```
<a href="/login" class="login-link" hidden>Login</a>
```

Make a getter that selects the link with `querySelector`. You'll need to reference the element later to show or hide it based on the response from the `api/session` endpoint.

```
get $login(): Element {
    return this.shadowRoot.querySelector('.login-link') a\
s Element;
}
```

SessionService

We need a way to determine if the login link should be shown or hidden based on if the user currently has a session. That would mean the user has logged in and is authenticated.

We haven't made any services yet, so let's make a space for all services to be stored in the application source code. Make a new directory named `service` in `packages/client/src`.

```
mkdir packages/client/src/service
```

Populate the service directory with a new file named `session.ts`.

```
touch packages/client/src/service/session.ts
```

Since we haven't submitted a request to `/api/login`, the only response `/api/session` will give is a status code of 401, meaning

unauthenticated. When the user has a session, `/api/session` returns a status code of 200, meaning success. These status codes are meaningless to the developer building out `AppHeader`. The developer can't derive any meaning from 200 and 401, so establish a new contract between `SessionService` and `AppHeader` before proceeding.

In `session.ts` declare a new `enum` named `SESSION` that stores the state for user sessions: `OPEN` (when the endpoint responds with 200) or `CLOSED` (when the endpoint responds with 401). Export a new `type` named `SessionResponse` that contains a property named `session` that can either be `SESSION.OPEN` or `SESSION.CLOSED`. This `SessionResponse` will be returned somewhere in the session service for `AppHeader`, it is the contract between service and UI element.

```
export enum SESSION {
  OPEN,
  CLOSED,
}

export type SessionResponse = {
  session: SESSION.OPEN | SESSION.CLOSED;
};
```

Declare and export a new `class` named `SessionService`. Set the path for the REST API endpoint on a `private` property named `path` and establish the `constructor`.

```
export class SessionService {
  private path = "/api/session";
  constructor() {}
}
```

Establish a new method on `SessionService` named `getSession`, which should return `Promise` with a `SessionResponse` that could

be used in `AppHeader`. Use `fetch` to make a `GET` request on the API endpoint stored on `this.path`. The response from the endpoint is accessible via a call to `then`. Pass the status of the response (`res.status`) to another new method named `processSession`.

```
getSession(): Promise<SessionResponse> {
  return fetch(this.path, {
    method: 'GET',
  }).then((res) => this.processSession(res.status));
}
```

In `processSession`, serialize the `status` to either `SESSION.OPEN` or `SESSION.CLOSED` based on if the endpoint responds with a status of `200`.

```
processSession(status: number): SessionResponse {
  return {
    session: status === 200 ? SESSION.OPEN : SESSION.CLOS\
ED,
  };
}
```

When you are done, import `SESSION` and `SessionService` into `packages/client/src/component/header/AppHeader.ts`.

```
import { SESSION, SessionService } from "./../../service/\
session";
```

Establish a new instance of `SessionService` locally in `AppHeader.ts`.

```
const sessionService = new SessionService();
```

Declare a new `connectedCallback` on the `AppHeader class`. First, check the user isn't on the `LoginView`, which will be at `/login`. Use `window.location.pathname` to check if the `pathname` includes `login`. If it doesn't, call `getSession` and depending on the response, remove the attribute from the element reference by the `this.$login` getter.

```
connectedCallback() {
  if (!window.location.pathname.includes('login')) {
    sessionService.getSession().then((status) => {
      if (status.session === SESSION.CLOSED) {
        this.$login.removeAttribute('hidden');
      }
    });
  }
}
```

To complete the basic functionality of `AppHeader`, we need to direct the user to `MainView` when they click the logo. Import `Listen` from `@in/common` to handle this use case.

```
import { attachShadow, html, css, Component, Listen } fro\
m "@in/common";
```

Decorate a new method named `onLogoClick` with the `Listen` decorator and navigate the browser to the root, which should display the `MainView` component.

```
@Listen('click', '.icon')
onLogoClick() {
  document.location.href = '/';
}
```

In this section, you learned how to reuse components from a UI library and application components directory in the context of a

view component named `MainView`. You abstracted http requests to a service layer named `SessionService`. Calling methods on `SessionService`, you were able to hide the link based on the condition of the user session. In the next chapter, you'll develop the `LoginView` which allows the user to authenticate with the backend and establish an active session. When that happens, you should observe the link is hidden. Until then, you'll develop parts of the `MainView` using a strict separation of concerns, continuing with `CookieFooter` next.

Cookie Footer

The landing page has a requirement for a banner at the bottom of the page to display legal messaging for the end-user who has to accept or deny cookies per GDPR standards. In Figma, the mockup displays a footer with a message and two buttons. Once the user accepts or denies cookies, a request must be sent to the backend to save their preference in the database. To support this feature, you'll develop a UI component that displays the footer and service that makes HTTP POST requests to the `/api/cookies` REST endpoint.

This component's functionality resembles `AppHeader` in some ways, although the service you'll build requires two requests: a `GET` request to determine whether or not to display the `CookieFooter` and a `POST` request to update the backend with the option the user chose by clicking on one of the buttons in `CookieFooter`. Otherwise, `CookieFooter` still requires some styling and a template, methods that handle click events and some initial logic that determines if the component should be displayed or hidden.

To get started, make a new directory in `packages/client/src/component` for footer components.

```
mkdir packages/client/src/component/footer
```

Create a new file in the footer directory named CookieFooter.ts.

```
touch packages/client/src/component/footer/CookieFooter.ts
```

`CookieFooter` requires a template and styling and those are self-contained within Shadow DOM, which only an autonomous custom element can provide. Import the necessary decorator and `function` from `@in/common` in `CookieFooter.ts` to get started.

```
import { attachShadow, html, css, Component } from "@in/c\
ommon";
```

Scaffold two `const`, one for styling and the other, the component's template.

```
const styles = css``;
const shadowTemplate = html``;
```

Use the `@Component` decorator to establish the tag name, styling, and template for a new `class` named `CookieFooter` that extends `HTMLElement`. Instantiate the `class` with a new `constructor` that calls `attachShadow`.

```
@Component({
  selector: "cookie-footer",
  style: styles,
  template: shadowTemplate,
})
export class CookieFooter extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

To use the new `CookieFooter class`, you'll need to export it from `packages/client/src/component/index.ts`.

```
export { AppHeader } from "./header/AppHeader";
export { CookieFooter } from "./footer/CookieFooter";
```

In turn, export `CookieFooter` in the application entry point (`packages/client/src/index.ts`) so it gets built with the rest of the project.

```
export { ButtonComponent } from "@in/ui";
export { AppHeader, CookieFooter } from "./component";
```

Follow up by updating `MainView` with a reference to the new `CookieFooter` in that component template. This will enable you to develop `CookieFooter` and observe live updates in the browser.

Template

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root"></div>
  <cookie-footer></cookie-footer>
`;
```

The template for `CookieFooter` includes a paragraph of legal messaging, followed by two buttons. Place the buttons in a `HTMLDivElement` container for easier styling.

```
const shadowTemplate = html`
  <p class="message">
    We use cookies to personalize content and ads, to pro\
vide social media
    features and to analyse our traffic.
  </p>
  <div class="button-container">
    <button is="in-button" class="in-button secondary">De\
ny</button>
    <button is="in-button" class="in-button primary">Allo\
w</button>
  </div>
`;
```

Styling

To achieve the styling shown in the mockup, give `:host` flex box characteristics that separate the `HTMLParagraphElement` and `HTMLDivElement` on either side of the footer.

```
const styles = css`
  :host {
    display: flex;
    justify-content: space-between;
    height: 120px;
    padding-left: var(--margin-lg);
    padding-right: var(--margin-lg);
    border-top: var(--border-default);
    align-items: center;
  }
`;
```

Fix some issues with the buttons by setting `align-self` to `right`, giving the container `display: flex` and establishing the `height`.

```
.button-container {
  align-self: right;
  display: flex;
  height: 40px;
}
```

The buttons have a standard margin between them, so add that styling.

```
button {
  margin-left: var(--margin-sm);
}
```

Finally, style the content appropriately for mobile devices by adding a media query that targets any design below 720px.

```
@media (max-width: 720px) {
  :host {
    flex-direction: column;
    margin-bottom: var(--margin-lg);
  }
}
```

Handling Click Events

When the user clicks on a button, the component will call on a service method that doesn't exist yet. At the very least, make an empty callback for the event listener on each button named `updateCookiePermission`, which should take a boolean: `true` when the user clicks the **Accept** button or `false` when the user clicks the **Deny** button.

```
updateCookiePermission(allow: boolean) {}
```

Import the `@Listen` decorator from `@in/common` to attach the event listeners.

```
import { attachShadow, html, css, Component, Listen } fro\
m "@in/common";
```

Declare two new methods on the `CookieFooter class` for each click listener on both buttons. Target the buttons by class name in the second argument of `Listen` and supply methods that call `updateCookiePermission` with the appropriate value.

```
  @Listen('click', '.secondary')
  onDenyClick() {
    this.updateCookiePermission(false);
  }
  @Listen('click', '.primary')
  onAllowClick() {
    this.updateCookiePermission(true);
  }
```

The implementation of `CookieFooter` is nearly complete. All that remains are calls to methods on the `CookieService` that make HTTP requests to the `/api/cookies endpoint`.

CookieService

To start developing the `CookieService`, make a new file in `packages/client/src/service` named `cookies.ts`.

```
touch packages/client/src/service/cookies.ts
```

Establish the contract between `CookieService` and `CookieFooter` with a new type definition named `CookiePermission`, an `Object` with one property named `permission` that can be equal to one of two values on the `COOKIES enum`. Eventhough the client has this type safe contract, the `/api/cookies` endpoint responds with a similar JSON `Object`, although the `permission` is set to a `Boolean`. Make another type definition named `InternalCookiePermission` that describes this behavior of the `/api/cookies` endpoint. As the name indicates, this type definition will be used to strictly type the response from the endpoint, while `CookiePermission` will be what `CookieFooter` references on the `COOKIES enum` to set the permission.

```
type InternalCookiePermission = {
  permission: boolean;
};

export enum COOKIES {
  ACCEPT,
  DECLINE,
}

export type CookiePermission = {
  permission: COOKIES.ACCEPT | COOKIES.DECLINE;
};
```

Export the `CookieService class` from the file, while giving the `class` a `constructor` and `path` property set to the path of the `/api/cookies` endpoint.

```
export class CookieService {
  private path = "/api/cookies";
  constructor() {}
}
```

Declare a new method on `CookieService` named `getPermission` that uses `fetch` to make a `GET` request, then processes the JSON in the response, with another method named `processPermission`, which is yet to be declared. If the endpoint doesn't return a `200` status code, coerce the response into an `Object` that matches `InternalCookiePermission`. `getPermission` should return the `fetch`, which is described via the type definition `Promise<CookiePermission>`. `getPermission` will be used to determine whether to show or hide `CookieBanner` based on the response from the endpoint.

```
getPermission(): Promise<CookiePermission> {
  return fetch(this.path, {
    method: 'GET',
  })
    .then((res) => {
      if (res.status === 200) {
        return res.json();
      } else {
        return {
          permission: false,
        };
      }
    })
    .then((res: InternalCookiePermission) => this.process\
Permission(res));
}
```

Make another method named `processPermission` which takes an `InternalCookiePermission` in the first argument and returns `CookiePermission`. The purpose of this method is to provide a type-safe response for UI components consuming it, so transform the `Boolean` into one of two possibilities declared in the `COOKIES` enum: `COOKIES.ACCEPT` or `COOKIES.DECLINE`.

```
processPermission(model: InternalCookiePermission): Cooki\
ePermission {
  return {
    permission: model.permission === true ? COOKIES.ACCEP\
T : COOKIES.DECLINE,
  };
}
```

The user should be able to specify whether they want to accept or deny cookies on the site and their preference should be stored in a database. If UI components like `CookieBanner` have a contract

with `CookieService` in the form of the `CookiePermission`, this method should use that type definition in the first argument. The `/api/cookies` endpoint accepts a `POST` request in the format of `InternalCookiePermission`, so establish the `Boolean` based on the value of the `permission` property on `cookies`, passed through the first argument. Just like in `getPermission`, return `fetch`, this time specifying the `method` as `POST` because the app is sending JSON, set the appropriate `headers`, and set the `body` a JSONified version of `InternalCookiePermission` using `JSON.stringify`. Process the response in the same way as `getPermission`.

```
givePermission(cookies: CookiePermission): Promise<Cookie\
Permission> {
  const allow = cookies.permission === COOKIES.ACCEPT ? t\
rue : false;
  return fetch('/api/cookies', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ permission: allow }),
  })
    .then((res) => {
      if (res.status === 200) {
        return res.json();
      } else {
        return {
          permission: false,
        };
      }
    })
    .then((res: InternalCookiePermission) => this.process\
Permission(res));
}
```

That's all there is to the implementation of `CookieService`.

You established two methods that can be reused in any UI component. Next, you'll import `CookieService` into `CookieFooter` so the user can specify whether or not to accept cookies using the `givePermission` method. You'll also import `CookieService` into `MainView`, which determines whether to show or hide the `CookieFooter` with the `getPermission` method.

Update CookieFooter

Import `CookiesService` and `COOKIES` into `CookieFooter.ts` and instantiate a new instance of `CookieService` with the `new` operator.

```
import { COOKIES, CookieService } from "./../../service/c\
ookies";

const cookieService = new CookieService();
```

Update the event listener callbacks to use the new contract established in `CookieService`. Instead of `Boolean`, call `updateCookiePermission` with the appropriate values of the `enum` which is more verbose and easily human readable.

```
@Listen('click', '.secondary')
onDenyClick() {
  this.updateCookiePermission(COOKIES.DECLINE);
}
@Listen('click', '.primary')
onAllowClick() {
  this.updateCookiePermission(COOKIES.ACCEPT);
}
```

Update the first argument of `updateCookiePermission` to accept values from the `COOKIES enum`, then call the `givePermission` method, passing the `allow` argument to the method. Listen for the API

endpoint response by calling `then`. If the user accepted cookies, set the `CookieFooter` element `hidden` attribute to `true`, otherwise remove the attribute to show the `CookieFooter`.

```
updateCookiePermission(allow: COOKIES.ACCEPT | COOKIES.DE\
CLINE) {
  cookieService
    .givePermission({
      permission: allow,
    })
    .then((cookies) => {
      if (cookies.permission === COOKIES.ACCEPT) {
        this.setAttribute('hidden', 'true');
      } else {
        this.removeAttribute('hidden');
      }
    });
}
```

Go ahead and test if clicking the "Accept" button hides the `CookieFooter`. If you reload the page, the `CookieFooter` will reappear. That's because you haven't included logic to show or hide the `CookieFooter` when the page loads. At any point if you wish to reset the database entry for cookie permission, stop the development server and implement one of two options: manually update the `cookies` property on `packages/server/data/db.json` or use `git restore` to reset the database to its default values. Before proceeding, update the database using one of those two methods and restart the development server with `yarn start:app` to restore the visibility of `CookieFooter`.

Update MainView

We have two options when architecting a solution for determining whether to show or hide the `CookieFooter` when the page first loads.

Regardless, there needs to be a `GET` request to `/api/cookies` to establish the permission the user set. We could keep the implementation solely in `CookieFooter` and have the component determine if it should show or hide itself. `CookieFooter` is part of a layout though in `MainView`, which also is concerned with the elements in the layout showing or hiding in the overall layout. When one element disappears from the layout, other elements may have to change their styling to cope with the change. For this reason, we're going to opt to keep the logic for showing and hiding the `CookieFooter` in `MainView`.

To get started refactoring `MainView`, make a new getter that selects the `CookieFooter` in the Shadow Root of `MainView`.

```
get $cookieFooter() {
  return this.shadowRoot.querySelector('cookie-footer');
}
```

Just like you did in `CookieFooter`, import `CookieService` and `COOKIES` from `cookies.ts` and instantiate the service.

```
import { COOKIES, CookieService } from "./../../service/c\
ookies";

const cookieService = new CookieService();
```

Establish a `connectedCallback` on `MainView` so you can reliably reference the `CookieFooter` in the template. In `connectedCallback` call `getPermission` on the `cookieService` and listen for a response from the API endpoint by calling `then`. Depending on the condition of the `CookiesPermission`, hide or show the `CookieFooter`.

```
connectedCallback() {
  cookieService.getPermission().then((cookies) => {
    if (cookies.permission === COOKIES.ACCEPT) {
      this.$cookieFooter.setAttribute("hidden", "true");
    } else {
      this.$cookieFooter.removeAttribute("hidden");
    }
  });
}
```

In the next chapter, you'll return to this `connectedCallback` to show a link on this landing page depending on if the user has a session. Currently the user can't even log in to establish a session, so we'll hold off on implementing this feature for now. Next, you'll code the remaining sections of the landing page, working between `MainView` and a new component that lazyloads images.

Landing Page Content

Add the following template to `MainView`, editing the `HTMLDivElement` with new sections that have boilerplate copy.

```
<div id="content-root">
  <div class="section">
    <div class="blurb half right">
      <h2>Your Last Contact List</h2>
      <p>
        Lorem ipsum dolor sit amet, consectetur adipiscin\
g elit. Quod eo
        liquidius faciet, si perspexerit rerum inter eas \
verborumne sit
        controversia.
      </p>
      <a href="/dashboard" class="cta dashboard-link">Vie\
```

```
13  w Contacts</a>
14      </div>
15    </div>
16    <div class="section">
17      <div class="blurb">
18        <h2>Turn Group Chats Into Live Events</h2>
19        <p>
20          Indicant pueri, in quibus ut in speculis natura c\
21  ernitur. Itaque ad
22          tempus ad Pisonem omnes. Quis hoc dicit? Philosop\
23  hi autem in suis
24          lectulis plerumque moriuntur. Videmus in quodam v\
25  olucrium genere non
26          nulla indicia pietatis, cognitionem, memoriam, in\
27   multis etiam desideria
28          videmus. Quid sequatur, quid repugnet, vident. At\
29  qui pugnantibus et
30          contrariis studiis consiliisque semper utens nihi\
31  l quieti videre, nihil
32          tranquilli potest.
33        </p>
34      </div>
35    </div>
36  </div>
```

Styling

Add styling for the link, being sure to include specific styling that displays an arrow after the words “View Contacts”.

```
a:link,
a:visited {
  font-family: var(--font-default);
  font-weight: var(--font-weight-default);
  font-size: var(--font-body-md);
  text-decoration: none;
  color: var(--color-black);
}
.cta:after {
  display: inline-block;
  margin-left: 4px;
  content: "\\25BA";
}
```

After you're done styling the link, add the `hidden` attribute to the DOM element, effectively hiding the call to action for now. This is intended functionality. The link should only be visible if the user has a session and we haven't implemented authentication yet.

```
<a href="/dashboard" class="cta dashboard-link" hidden>Vi\
ew Contacts</a>
```

Give the content container some flexbox styling so you can better visualize subsequent styling changes.

```
#content-root {
  display: flex;
  flex-direction: column;
}
```

According to the mockup, every section has some padding. The template also includes CSS classes on some elements that are intended to augment their size or position. Every text "blurb" has justified text. Style these elements accordingly, being sure to also provide styling for smaller screens.

```
.section {
  padding: var(--margin-lg);
}
.blurb {
  text-align: justify;
}
.half {
  width: 50%;
}
.right {
  float: right;
}

@media (max-width: 480px) {
  .half {
    width: 100%;
  }
  .third {
    width: 100%;
  }
  .right {
    float: left;
  }
}
```

Background

To begin, let's add the intended attributes to the template in `MainView` so we can develop against them in yet to be created `Background` custom elements. On the first section `HTMLDivElement`, set the `is` attribute to the tag name of the new customized built-in element: `in-bg`. Set the `background` attribute to the path for the background image.

```
<div
  is="in-bg"
  class="section"
  background="/style/asset/timon-studler-BIk2ANMmNz4-unsp\
lash.jpg"
```

Do the same for the second section, setting the `background` attribute to another image.

```
<div
  is="in-bg"
  class="section"
  background="/style/asset/valiant-made-zBkVp3E2CnE-unspl\
ash.jpg"
```

Make a new directory named `background` at `packages/client/src/component`

```
mkdir packages/client/src/component/background
```

Create a new file in this directory named `Background.ts`

```
touch packages/client/src/component/background/Background\
.ts
```

Import the `@Component` decorator for this customized built-in element.

```
import { Component } from "@in/common";
```

Declare the new `class` named `Background`, ensuring it extends `HTMLDivElement`. Add the mandatory `constructor` with a call to `super()`. using the `@Component` decorator, set the `selector` to `in-bg` and use the `custom` property to set the configuration for the `CustomElementRegistry`.

```
@Component({
  selector: "in-bg",
  custom: {
    extends: "div",
  },
})
export class Background extends HTMLDivElement {
  constructor() {
    super();
  }
}
```

Export `Background` from `packages/client/src/component/index.ts`.

```
export { AppHeader } from "./header/Header";
export { CookieFooter } from "./footer/CookieFooter";
export { Background } from "./background/Background";
```

Similarly, export `Background` from `packages/client/src/index.ts`.

```
export { Background, AppHeader, CookieFooter } from "./co\
mponent";
```

On the `Background class`, establish a static getter to listen for attribute changes on the `background` attribute.

```
static get observedAttributes() {
    return ['background'];
}
```

Set up the `attributeChangedCallback` to call a new method named `setBackground` with the value of the `background` attribute.

```
attributeChangedCallback(name, prev, next) {
    switch (name) {
        case 'background':
        this.setBackground(next);
        break;
    }
}
```

Declare the `setBackground` method, which sets various CSS styles on the element. Pass the `path` to `this.style.background`, being sure to wrap the path in the required `url()` syntax. Next, set the background position and size, which have to be set after the background image in order for the browser to reliably draw the image.

```
setBackground(path: string) {
    this.style.background = `url(${path})`;
    this.style.backgroundPosition = 'center center';
    this.style.backgroundSize = 'cover';
}
```

Update Styling

Update `MainView` styles with the appropriate styling for each section with a background image.

```
[is="in-bg"] {
  width: calc(100% - (var(--margin-lg) * 2));
  min-height: 480px;
}
```

Correct the padding on the last section that is getting cut off behind the `CookieFooter`.

```
.section:last-child {
  padding-bottom: 120px;
}
```

Establish a new style declaration for any element you want to make the text white.

```
.light {
  color: var(--color-white);
}
```

Add the `light` class to the second section.

```
<div
  is="in-bg"
  class="section light"
  background="/style/asset/valiant-made-zBkVp3E2CnE-unspl\
ash.jpg"
```

In this section, you used a customized built-in element to lazyload background images of the two marketing sections on the landing page. Check the layout looks close to the mockups in Figma before proceeding to the next section. If anything seems out of place, review the steps in this chapter and fix any issues.

Summary

In this chapter, you developed a landing page for the application with custom elements. You created a separation of concerns between UI components, views, and services, which should help the application scale. In the next chapter, you'll continue this trend, coding more services for authentication and navigation that can be used in the context of any view or component.

If you chose not to abstract the styling and templates into separate `const` in each of these components, that may come back to haunt you in Chapter Ten where you make each application component compatible with server-side rendering (SSR). SSR will be extremely valuable for the landing page, which includes content that could drive visitors to the application. As of now, we can't guarantee search engines can crawl the page. The relevant page content is hidden in Shadow DOM. In Chapter Ten, you'll overcome this limitation by implementing Declarative Shadow DOM templates for each component you worked on in this chapter.

You may have noticed that so far, you've only used one component from the UI library (`ButtonComponent`). All that hard work from previous chapters will pay off in the next chapter when you'll develop the `LoginView` and `DashboardView` with the rest of the components exported from the `@in/ui` package. This should hopefully accelerate development, while keeping you focused on implementing the features at hand. Before working on the login and dashboard components in the next chapter, you'll develop a client-side router that can navigate between all three views, including the `MainView` you developed in this chapter.

Routing Custom Elements in a SPA

Chapter Nine

Routing Custom Elements in a SPA

Single-page applications (SPA) are web applications that load only one HTML document. This approach to development can be advantageous for several reasons. Only one HTML file needs to be deployed to production instead of many. Resources don't have to be loaded over and over again for the end-user. SPA continuously updates the existing `document.body` as the user navigates, reusing existing CSS and JavaScript resources. SPA have some trade-offs, namely the inability to be optimized for search engines (SEO), the requirement to implement some sort of client-side routing mechanism, and the potential for an increased JavaScript bundle size. The advantages of SPA can result in the degradation of performance for the end-user.

Luckily, we're implementing custom elements with a micro-library (`@in/common`) that should have a minimal negative impact on performance. SEO remains a problem in the SPA we're developing because HTML elements aren't appended to the `document.body` until the application bundle loads. To be more performant and SEO compliant, the HTML elements on the landing page could be server-side rendered. We'll cover server-side rendering in Chapter Ten.

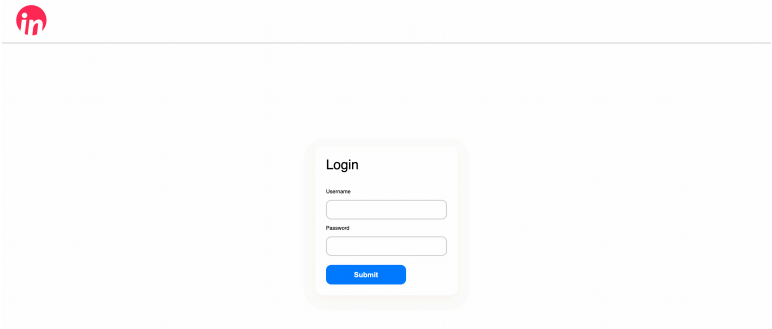
With SEO and performance concerns out of that way, that just leaves the problem of routing. Since the build tooling for this application doesn't deploy separate HTML files for each view, we

need to come up with a way for the application to navigate from `MainView` to the yet to be developed `LoginView`. Once authenticated in the `LoginView`, the application should route to the `DashboardView`. This kind of navigation requires a client-side router.

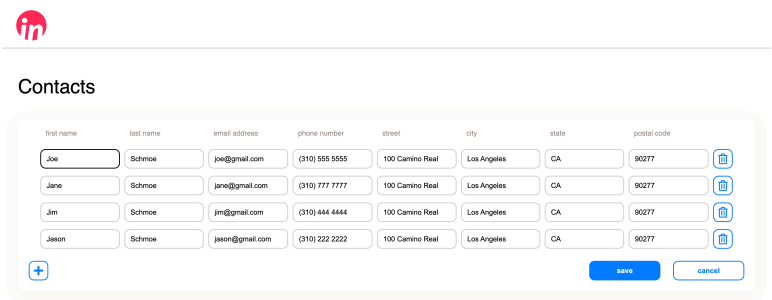
Major JavaScript frameworks often include or endorse a client-side router. It could make a lot of sense to adopt a JavaScript framework and exclusively relegate custom elements to UI library that is framework-agnostic, although we've already chosen to use custom elements in the view layer of the application. That means we need a router that can interpret custom elements and replace the view content with the template or instance of a custom element like `MainView`. This task may seem daunting if you've never coded a router before, so it's hard to know where to begin. Let's start with some characteristics observable in many framework routers.

- Configurable in some way, usually through a manifest of routes
- Creates and destroys HTML content at a specified root DOM node, i.e. `<body>` or `<div id="root">`
- Parses the path in the browser address bar and loads a specific route based on the path
- Manipulates browser history state so user can still use the "Back" button

In this chapter, you'll develop a minimal client-side router that does all of these things. `Router` will be a `class` that can be instantiated with the selector for the root node (`main-view`) and configuration for each route. When the browser navigates to a path that matches a particular route, the router will instantiate the view component associated with that route and append the element to the container `HTMLDivElement` available on the `document`: `<div id="root">`. When you finish coding the router, you'll have a simple implementation that can be expanded upon to allow for hashed routes, parsing query params, reacting to "pop" events, and more.



While developing `MainView` in the last chapter, you coded new custom elements. In this chapter, you'll reuse `AppHeader` and several components exported from `@in/ui` while developing the `LoginView` and `DashboardView`. Some of the logic for these views can even be lifted from the `stories.js` files in Storybook, which provide examples of how to use `TextInputComponent` in the context of a form or `TableComponent`. The main difference between the Storybook examples and the application is that the components in the application need to interact with a backend that exposes a REST API. During development of the `LoginView` and `DashboardView`, you'll make several other services that support making HTTP requests to REST API endpoints.



What you will build

- `LoginView` which displays the global header and login interface
- `DashboardView` which displays the global header and table
- `Router` which handles client-side routing between main, login and dashboard views
- `RouteConfig` which configures a single route
- `LocationService` which gets the current path and navigates the user to another path
- `ContactService` which fetches and modifies a user's contact list

What you will learn

- How to code a client-side router for single page application built with custom elements
- How to handle authentication for a client-side application
- How to reuse custom elements from a UI library during application development

Acceptance criteria

Getting started

Navigate to the project directory:

```
1 cd fullstack-web-components
```

Checkout a new branch named chapter-9 from where you left off at the end of chapter 8.

```
git checkout -b chapter-9
```

Routing

In this section, you'll develop the `Router class`, which is responsible for instantiating and appending a view to the root element on the `document`. After coding the `Router` implementation, you'll test the router works by scaffolding the two view components that should be displayed with the user lands on `/login` and `/dashboard`. When the user clicks the "login" link on the landing page, the `Router` should navigate to `/login` and display the template of the `LoginView` autonomous custom element. Similarly, when the browser navigates to `/dashboard`, the `DashboardView` template should appear.

`Router` should be configured in the application logic somewhere and that configuration should be injected into the `Router class` when it is instantiated. By separating the configuration into a separate file, we're setting ourselves up for success in Chapter Ten, where the server-side renderer will reference the same router configuration to configure each route on the server.

To get started, make a new directory for the router.

```
mkdir packages/client/src/router
```

Create a new file in the router directory named `index.ts`.

```
touch packages/client/src/router/index.ts
```

For starters in `index.ts`, let's consider the configuration of each route. Export a new `interface` named `Route` from `index.ts` and start defining the types of the properties on `Route`. For the `Router` to match a route based on the path in the address bar, we'll need to

store the path on the `Route`. This way, `Router` will be able to check if a route.path that matches the path in the address bar exists. Each `Route` should have a view component that displays a template that will be appended to the root element. `MainView` is an example of one of these view components. The component could be type defined to a `string` and we can use the tag name custom element that would be passed to `document.createElement` before the instance of the view is appended to DOM. Finally, each route should update the `title` tag in the `document.head`, so add `title` as another property on `Route`.

```
export interface Route {
  path: string;
  tag: string;
  title?: string;
}
```

Routes

Make a new file named `routes.ts` in `packages/client/src`. This file will store the router configuration.

```
touch packages/client/src/routes.ts
```

```
import { Route } from "./router";
```

In `routes.ts`, export a `const` named `routes` and make that equal to an `Array<Route>`. Define the configuration for each `Route` in the `Array`, passing each route, the path, tag name of the view component, and title that should be displayed in the browser window.

```
export const routes: Array<Route> = [
  {
    path: "/",
    tag: "main-view",
    title: "In",
  },
  {
    path: "/login",
    tag: "login-view",
    title: "Login",
  },
  {
    path: "/dashboard",
    tag: "dashboard-view",
    title: "Contact Dashboard",
  },
];
```

Router

In `packages/client/src/router/index.ts`, export a new `class` named `Router`. Declare two properties on `Router` named `rootElement` and `routes`. `rootElement` should be type defined as `Element`. Eventually, the root element all routes are appended to will be referenced here. `routes` should be type defined as `Array<Route>`. Add a `constructor` to the `Router` `class` and instantiate the `class` with two arguments: `root`: a selector that can be passed to `document.querySelector`, and `routes`, the `Array<Route>` you just created in the `routes` configuration.

```
export class Router {
  rootElement: Element;
  routes: Array<Route>;
  constructor(root: string, routes: Array<Route>) {}
}
```

Before proceeding with setting the two properties, add some error handling if the root node doesn't exist or if `Router` is instantiated without `Array<Route>`.

```
constructor(root: string, routes: Route[]) {
  if (document.querySelector(root) === null) {
    console.error(`[Router] Root element '${root}' does n\
ot exist.`);
  }
  if (!routes) {
    console.error(`[Router] initialized without any route\
s.`);
  }
}
```

Set the `rootElement` and `routes` properties with the values of the two arguments. At the end of the `constructor`, call a method that doesn't exist yet: `init()`.

```
constructor(root: string, routes: Route[]) {
  if (document.querySelector(root) === null) {
    console.error(`[Router] Root element '${root}' does n\
ot exist.`);
  }
  if (!routes) {
    console.error(`[Router] initialized without any route\
s.`);
  }
  this.rootElement = document.querySelector(root);
```

```
    this.routes = routes;
    this.init();
  }
```

All `init` will do for now is call another method `navigate`, which takes the path of the `Route` in the first argument. Since the page has just loaded and the `Router` initialized, use the current path as seen in the address bar (`window.location.pathname`) to call `navigate`.

```
init() {
  this.navigate(window.location.pathname);
}
```

In the `navigate` method, create a new `const` that should be equal to a `Route`. To find the `Route` that matches the `path` passed into the first argument of `navigate`, you'll need another method called `matchRoute`.

```
navigate(path: string) {
  const route = this.matchRoute(path);
}
```

Declare `matchRoute` and make it return an algorithm that loops through `this.routes` and finds a `Route` where the `path` property on the `Route` matches the `path` passed into the first argument of `matchRoute`. While using `Array.prototype.find`, if there isn't a match, `find` should return `undefined`.

```
matchRoute(path: string): Route | undefined {
  return this.routes.find((route) => route.path === path);
}
```

Account for the use case where there isn't a match and throw an error in the console, then `return`. This ensures the router won't

remove any content before appending the new view. That logic should happen in another method called `resolve`. Pass the `route` into the new `resolve` method.

```
navigate(path: string) {
  const route = this.matchRoute(path);
  if (!route) {
    console.error(`[Router] Route '${path}' does not exis\
t.`);
    return;
  }
  this.resolve(route);
}
```

`resolve` has a valid `Route` to display, so let's use the remaining properties on the `Route` (`tag` and `title`) to display the view component at that route. Create a new `const` named `component`, making it equal to `document.createElement`, while passing in the `route.tag`. Set the `document.title` with the value of `route.title`. Finally, wipe out the content of `rootElement` and append the `component`.

```
resolve(route: Route) {
  const component: HTMLElement = document.createElement(r\
oute.tag);

  if (route.title) {
    document.title = route.title;
  }

  this.rootElement.innerHTML = '';
  this.rootElement.appendChild(component);
}
```

We're finished with the basic implementation of a router that appends custom elements when a route matches the path in the

browser address bar. To use this new `Router`, we have to refactor the entry point of the application. Previously, we manually get the content of the root element with the `MainView`. This should now happen dynamically with `Router`, so remove the following lines from `packages/client/src/index.ts`.

```
const mainView = document.createElement("main-view");
document.querySelector("#root").appendChild(mainView);
```

Import the `Router` and `routes` into the same file. Instantiate the `Router`, passing in the selector for the root element (`#root`) and the `Array<Route>` stored on `routes`. Export all three from the application bundle.

```
import { routes } from "./routes";
import { Router } from "./router";

const router = new Router("#root", routes);

export { routes, router, Router };
```

To test the new `Router` works, you'll need more than one view component. Scaffold the login and dashboard view components next, declaring temporary templates on both that allow you to test the router.

Scaffold LoginView

Make a new directory named login at `packages/client/src/view` that will store the file that exports the login view custom element.

```
mkdir packages/client/src/view/login
```

Create a new file named `LoginView.ts` in the login directory.

```
touch packages/client/src/view/login/LoginView.ts
```

Scaffold this new component that will be used to display the login view much the same way you scaffolded `MainView` in the previous chapter. Import all the necessary functionality from `@in/common` to develop an autonomous custom element.

```
import { attachShadow, css, html, Component } from "@in/c\
ommon";
```

Declare `const` for the styling and template. Give the `shadowTemplate` a temporary template that includes two links. This will help validate the router works.

```
const styles = css``;
const shadowTemplate = html`
  <h1>Login</h1>
  <a href="/">Landing Page</a>
  <a href="/dashboard">Dashboard</a>
`;
```

Export a new `class` named `LoginView` that extends `HTMLElement`, being sure to call `attachShadow` and `super` in the `constructor`. Decorate the `class` with the `@Component` decorator, establishing `login-view` as the `selector` and passing the `styles` and `shadowTemplate` to their respective properties.

```
@Component({
  selector: "login-view",
  style: styles,
  template: shadowTemplate,
})
export class LoginView extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

Scaffold DashboardView

Repeat the process for the dashboard view. Make a new directory named `dashboard` at `packages/client/src/view`.

```
mkdir packages/client/src/view/dashboard
```

Create a new file named `DashboardView.ts` in the `dashboard` directory.

```
touch packages/client/src/view/dashboard/DashboardView.ts
```

`DashboardView.ts` should appear like below when you are finished scaffolding the file.

```
import { attachShadow, css, html, Component } from "@in/c\
ommon";

const styles = css``;
const shadowTemplate = html` <h1>Dashboard</h1>
  <a href="/">Landing Page</a>
  <a href="/login">Login</a>`;

@Component({
  selector: "dashboard-view",
  style: styles,
  template: shadowTemplate,
})
export class DashboardView extends HTMLElement {
  constructor() {
    super();
    attachShadow(this);
  }
}
```

Export all three views in `packages/client/src/view/index.ts`.

```
export { MainView } from "./main/MainView";
export { LoginView } from "./login/LoginView";
export { DashboardView } from "./dashboard/DashboardView";
```

Update `packages/client/src/index.ts` with the new exports.

```
export { DashboardView, LoginView, MainView } from "./vie\
w";
```

Now `LoginView` and `DashboardView` are properly scaffolded and `MainView` exists from the previous chapter, test the new router works. Start at http://localhost:4443 and click the "Login" link in `AppHeader`. You should notice the browser loads the `LoginView`.

Click on the link for the "Dashboard" in the `LoginView` and noticed `DashboardView` loads. Finally, check the router works correctly by clicking the "Landing Page" link in the `DashboardView`. Once you have validated that the router works, you're ready to provide more content to `LoginView` and `DashboardView` in the next sections of this chapter.

Login View

The login view displays the `AppHeader` along with a `CardComponent` that includes the form elements ncessary for a user to sign into the application. Luckily, most of the UI in this view has already been developed or scaffolded for us. We can just use `AppHeader` because it's exported from the application bundle. The same goes for any component exported from `@in/ui`. Webpack is currently bundling the application and has included all the components from`@in/ui`in the bundle by default. In Chapter Three, you developed most of the form in the context of Storybook, so we can lift the example from there and apply it to the`LoginViewtemplateandclass‘.

To get started, open `LoginView.ts` and navigate to http://localhost:4443/login with the development server running. Deprecate the temporary template from the last section you used to test the router and instead provide a template that includes the `AppHeader`, a new `HTMLDivElement` with the `id` attribute set to `content-root`. This mirrors the template in `MainView.ts`. By standardizing the way these templates are structured, it will be way easier to refactor these components for server-side rendering in the next chapter. Add an instance of `CardComponent` to the content area, ensuring the element has a `max-width` of `320px`. If you lifted the `CardComponent` from Storybook, it will come with this styling inline. Add a heading element with the title `Login`.

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root">
    <in-card style="max-width: 320px">
      <h4 slot="header">Login</h4>
    </in-card>
  </div>
`;
```

The `HTMLFormElement` can be exactly the same as the story, with one notable exception. The credientials for the development environment you'll use to log in are only five characters long, so if you copied the template from the story, adjust the `minlength` on each `TextInputComponent` to `5`.

```
<form
    name="foo"
    slot="content"
>
  <fieldset>
      <legend>Login Form</legend>
      <label for="username">Username</label>
      <in-textinput
      type="text"
      id="username"
      name="username"
      required
      minlength="5"
      class="form-control"
      ></in-textinput>
      <label for="password">Password</label>
      <in-textinput
      type="password"
      id="password"
      name="password"
```

```
      required
      minlength="5"
      class="form-control"
      ></in-textinput>
      <button class="primary submit form-button" is="in-b\
utton">Submit</button>
</form>
```

Styling

Next, let's define the styling for the `LoginView`. Give the `LoginView` some basic styling, which should properly display the `AppHeader` and content with flexbox. Adjust the content area so it displays in the center of the browser window.

```
const styles = css`
  :host {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    height: 100vh;
  }
  #content-root {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translateX(-50%) translateY(-50%);
    min-width: 320px;
  }
`;
```

Style the heading per the mockups.

```
h4 {
  font-family: var(--font-default);
  font-weight: var(--font-weight-default);
  font-size: var(--font-headline-line-height-sm);
  margin-block-start: 0em;
  margin-block-end: 0em;
  margin-top: var(--margin-lg);
  margin-bottom: var(--margin-lg);
}
```

Override the browser defaults for `fieldset`, which shouldn't recieve any styling according to the mockups.

```
fieldset {
  border: 0 none;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
  padding-block-start: 0em;
  padding-inline-start: 0em;
  padding-inline-end: 0em;
  padding-block-end: 0em;
}
```

Override the browser defaults for `legend`, which should be part of the layout for accessibility reasons, but can remain visually hidden.

```
legend {
  visibility: hidden;
  height: 0px;
}
```

Style the `label` elements.

```
label {
  display: block;
  font-family: var(--font-default);
  font-size: var(--font-body-sm);
  margin-top: var(--margin-sm);
  margin-bottom: var(--margin-sm);
}
```

Provide a simple fix for the spacing in between the form inputs and the button.

```
button {
  margin-top: var(--margin-md);
}
```

Form Logic

Back in Chapter Three, we devised a way to handle validation using an `Object` with keys that were equal to the `name` attributes on each `TextInputComponent`. Each key stores a `Validator`, a custom schema used to provide context to `ElementInternals` and the `TextInputComponent`, which should display a validation message. Internally, `TextInputComponent` interprets a `Validator` set on the element's `$validator` property, referencing the `Validator` on this property for the validation message. Copy the `validators Object` from Storybook, adjusting the minimum length messaging to account for 5 characters. Set the `Object` on a property of `LoginView` named `validators`.

```
export class LoginView extends HTMLElement {
  validators = {
    username: {
      validations: [
        {
          flag: { valueMissing: true },
          message: 'Error: Required, please enter a usern\
ame.',
          condition: (input) => input.required && input.v\
alue.length <= 0,
        },
        {
          flag: { tooShort: true },
          message:
            'Error: Minimum length not met, please supply\
 a value with at least 5 characters.',
          condition: (input) =>
            input.minLength && input.value.length < input\
.minLength,
        },
      ],
    },
    password: {
      validations: [
        {
          flag: { valueMissing: true },
          message: 'Error: Required, please enter a passw\
ord.',
          condition: (input) => input.required && input.v\
alue.length <= 0,
        },
        {
          flag: { tooShort: true },
          message:
            'Error: Minimum length not met, please supply\
```

```
 a value with at least 5 characters.',
          condition: (input) =>
            input.minLength && input.value.length < input\
.minLength,
        },
        {
          flag: { patternMismatch: true },
          message:
            'Please use at least one uppercase, uppercase\
 letter, special character, and number.',
          condition: (input) =>
            input.pattern &&
            input.value.match(new RegExp(input.pattern)) \
=== null,
        },
      ],
    },
  };
```

Declare a new `connectedCallback`. In this method, loop over the properties on `this.validators` and set the `$validator` property on each `TextInputComponent` with the value of the matching `Validator`.

```
connectedCallback() {
    for (let prop in this.validators) {
        (this.shadowRoot.querySelector(`[name="${prop}"]`\
) as any).$validator =
        this.validators[prop];
    }
}
```

Validation should happen automatically when the user blurs any of the `TextInputComponent`, but we'll need some special handlers to kick off validation when the user clicks the "Submit" button. To

attach an event listener to the button in the template, import the `@Listen` decorator from `@in/common`.

```
import { attachShadow, css, html, Component, Listen } fro\
m "@in/common";
```

Add a new method to `LoginView` named `onButtonClick` and attach the `@Listen` decorator to the method, being sure to listen for `click` and selecting the button by its CSS class name. Call `ev.preventDefault()` to prevent the default form submission the user agent will attempt to trigger. Instead, we want to handle validation and submission in the context of the `LoginView class`. Make a new method named `onValidate` and pass the `Event` to this method.

```
@Listen('click', '.form-button')
onButtonClick(ev) {
  ev.preventDefault();
  this.onValidate(ev);
}
```

Declare the `onValidate` method, which should also be a callback for the `validate Event` sent by each `TextInputComponent`.

```
@Listen('validate')
onValidate(ev: Event): void {

}
```

In the `onValidate` method, loop over `this.validators`, pushing the validity of each form input into a new `Array` named `validations`.

```
const validations = [];
for (let prop in this.validators) {
  validations.push(
    (this.shadowRoot.querySelector(`[name="${prop}"]`) as\
 any).validity?.valid
  );
}
```

If any item in the `validations Array` is falsy, the form is invalid, otherwise, call a new method named `onSubmit` when the form is valid. Console messages may help here to diagnose any issues.

```
if (validations.filter((val) => val == false).length) {
  console.warn("INVALID");
} else {
  console.log("VALID");
  this.onSubmit();
}
```

The complete `onValidate` method should appear like below when you are done coding it.

```
@Listen('validate')
onValidate(ev: Event): void {
    const validations = [];
    for (let prop in this.validators) {
        validations.push(
        (this.shadowRoot.querySelector(`[name="${prop}"]`)\
) as any).validity
            ?.valid
        );
    }
    if (validations.filter((val) => val == false).length)\
 {
        console.warn('INVALID');
```

```
    } else {
        console.log('VALID');
        this.onSubmit();
    }
}
```

Declare a new method named `onSubmit`.

```
@Listen('submit')
onSubmit(): void {

}
```

In the `onSubmit` method, we need to prepare the body of the `POST` request that will be made to the `/api/login` API endpoint. Loop over each form control and populate the request body with the value of the username and password `TextInputComponent`.

```
const request = {};
Array.from(this.shadowRoot.querySelectorAll(".form-contro\
l")).forEach(
  (control: any) => {
    request[control.id] = control.value;
  }
);
```

LoginService

Just like in `MainView`, we're going to abstract the http requests to a service known as `LoginService`. Start off by establishing the contract between service and UI component by declaring the type definitions for `LoginRequest` and `LoginResponse`. A `POST` request to `/api/login` requires a JSON `Object` in the body of the request.

That `Object` contains two properties: `username` and `password`. Both are typed `string`. If the user can authenticate with the correct username and password, the endpoint responds with a status code of `200` (Success), otherwise `401` (Unauthorized). Declare a type definition for `InternalLoginResponse` that satisfies the status code. These status codes are hard to understand for the consumer of the `LoginService`. We already have an `enum` that describes the condition based on the response: `SESSION`. If the response from `/api/login` is successful, the session is `OPEN`, otherwise `CLOSED`. Define the type of the response as `LoginResponse` and add a property named `session` that can either be `SESSION.OPEN | SESSION.CLOSED`.

```
import { SESSION } from "./session";

export type LoginRequest = {
  username: string;
  password: string;
};

export type InternalLoginResponse = {
  status: number;
};

export type LoginResponse = {
  session: SESSION.OPEN | SESSION.CLOSED;
};
```

Export a new `class` named `LoginService` and instantiate the `class` with a property named `path` equal to the path of the API endpoint at `/api/auth`.

```
export class LoginService {
  private path = "/api/auth";
  constructor() {}
}
```

The `login` method takes a single argument: the body of the request formed in the `LoginView`, type defined as `LoginRequest`. This method returns a `Promise<LoginResponse>` because the method returns a call to `fetch`. Pass `this.path` to the first argument of `fetch`, then the configuration `Object` in the second argument that contains the `method` set to `POST`, the necessary `headers`, and finally the `body` set to the JSON strigified `request`. Use `then` to listen for the HTTP response from the endpoint and process that response with a new method named `processLoginResponse`.

```
login(request: LoginRequest): Promise<LoginResponse> {
  return fetch(this.path, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(request),
  }).then((res) => {
    return this.processLoginResponse(res);
  });
}
```

The `processLoginResponse` method serializes the status of the response on `InternalLoginResponse` to a human readable format defined by `LoginResponse`.

```
processLoginResponse(res: InternalLoginResponse): LoginRe\
sponse {
  return {
    session: res.status === 200 ? SESSION.OPEN : SESSION.\
CLOSED,
  };
}
```

Now that you have a method that calls the `/api/login` endpoint, import the `LoginService` class into `LoginView.ts` and instantiate the service.

```
import { LoginRequest, LoginService } from "../../service\
/login";

const loginService = new LoginService();
```

In the `onSubmit` method, define the type of the `request Object` as `LoginRequest`.

```
const request: LoginRequest = {
  username: "",
  password: "",
};
```

After `request` is populated with properties, call `loginService.login`, passing in the `request`. Listen for the `LoginResponse` using `then`. If the `session` property on `res` equals `SESSION.OPEN`, you'll need some logic here to navigate the browser to `/dashboard`.

```
loginService.login(request).then((res) => {
  // will navigate here on success
});
```

Test the endpoint is working by filling out the form with the username and password fields set to `super`. If you make a change after authenticating and the browser reloads, you will have to authenticate again with `super` to establish an active session.

To navigate the user, we want to update browser history (primarily in case the user presses the "back" button), but we also need a way to change the path in the browser address bar. It would get messy writing calls all over the codebase to update history state and then change the path in the address bar, so let's wrap this functionality in another service.

Make a new file named `location.ts` at `packages/client/src/service/location.ts`.

```
touch packages/client/src/service/location.ts
```

In `location.ts` export a new `class` named `LocationService`. Expose a single method on the service for now named `navigate`. This method should take the path you want to navigate the browser to, i.e. `login` or `dashboard`. Call `window.history.replaceState`, passing in the new path to the third argument. This enables us to store state in browser history if we so desired, like the component tag name that should be loaded at this route. Next, call `window.location.replace` to change the path in the address bar.

```
export class LocationService {
  constructor() {}
  navigate(route: string) {
    window.history.replaceState({}, "", `${window.locatio\
n.origin}/${route}`);
    window.location.replace(`${window.location.origin}/${\
route}`);
  }
}
```

Import `SESSION` and `LocationService` into `LoginView.ts`.

```
import { SESSION } from "../../service/session";
import { LocationService } from "../../service/location";

const locationService = new LocationService();
```

In the `onSubmit` method, check the condition of the `LoginResponse` and if `res.session`equals `SESSION.OPEN`, call `locationService.navigate` to navigate the browser to the dashboard route.

```
loginService.login(request).then((res) => {
  if (res.session === SESSION.OPEN) {
    locationService.navigate("dashboard");
  }
});
```

Go ahead and test the browser navigates to `/dashboard` once you can successfully authenticate with the username and password set to `super`. Any time you make a change and the browser reloads, you may have to authenticate again to test out features in the application that depends on an active session.

Dashboard View

Just like the login view, the dashboard displays the `AppHeader` and content area. `DashboardView` implementation should be less involved than `LoginView`. That's because the dashboard relies only on the response from a single API endpoint at `/apt/contacts`, which responds with a data model that can display the columns and rows of `TableComponent`. In Chapter Six, you developed the `TableCardComponent` using Storybook as your development environment. Several parts of the example found in `Table.stories.js` can be ported over to the implementation of `DashboardView`.

Template

Start by replacing the content of `shadowTemplate` with an instance of `AppHeader`, a `HTMLDivElement` with `id` set to `content-root`. In this `HTMLDivElement`, place a heading with the label `Contacts` and an instance of `TableCardComponent`. Set the mandatory `channel` attribute of `TableCardComponent` to the intended `BroadcastChannel` that will be used by the various components that make up `TableCardComponent`.

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root">
    <h1>Contacts</h1>
    <in-tablecard channel="dashboard-channel"></in-tablec\
ard>
  </div>
`;
```

Styling

The template of `DashboardView` is rather simple given all the complexity embedded within `AppHeader` and `TableCardComponent`. Focus momentarily on styling the view. Add some styling for the content area that gives the `HTMLDivElement` some `padding` and ensures the `width` always fills the container.

```
const styles = css`
  #content-root {
    display: block;
    margin: 0;
    padding: var(--padding-xl);
    width: 100%;
  }
`;
```

Add the appropriate styling for the `h1`.

```
h1 {
  font-family: var(--font-default);
  font-weight: var(--font-weight-default);
  font-size: var(--font-headline-line-height-md);
  margin-block-start: 0em;
  margin-block-end: 0em;
  margin-top: var(--margin-lg);
  margin-bottom: calc(var(--margin-lg) * 2);
}
```

Fix the padding on either side of `TableCardComponent` by setting the `width` to a calculation with `calc`.

```
in-tablecard {
  display: block;
  width: calc(100% - var(--padding-xl) * 2);
}
```

That's all there is to styling the view so turn your focus to fetching a data model for `TableComponent`.

ContactService

Make a new file for the contact service in `packages/client/src/service`.

```
touch packages/client/src/service/contacts.ts
```

Establish the contract between service and UI component. We already have a type defintion for the columns, but lack a type definition for the rows. Define the type of each row as `Contact`. Each property on `Contact` will display as a cell in the table. Define the type definition of the API endpoint response.

```
import { ColumnData } from "@in/ui";

export type Contact = {
  email: string;
  firstName: string;
  lastName: string;
  phoneNumber: string;
  street: string;
  city: string;
  region: string;
  postalCode: string;
};

export type ContactResponse = {
```

```
  columnData: ColumnData;
  rowData: Array<Contact>;
};
```

Export a new `class` named `ContactService`, instantiating the `class` with a property named `path` set to `/api/contacts`. Set the type defintion of another property named `channelName` which will store the name of the channel the service uses to communicate over `BroadcastChannel`. Instantiate the `class` with a mandatory argument of the same name, passing its value to `this.channelName`. Set another property named `channel` to a new instance of `BroadcastChannel`, passing in `channelName` to its constructor.

```
export class ContactService {
  private channel: BroadcastChannel;
  private channelName: string;
  private path = "/api/contacts";
  constructor(channelName: string) {
    this.channelName = channelName;
    this.channel = new BroadcastChannel(this.channelName);
  }
}
```

Make a method that handles the `GET` request to `/api/contacts` using `fetch`. Process the response with a new method named `processContacts`.

```
getContacts(): Promise<ContactResponse> {
  return fetch(this.path, {
    method: 'GET',
  })
  .then((res) => {
    if (res.status === 200) {
      return res.json();
    } else {
      return {
        columnData: [],
        rowData: [],
      };
    }
  })
  .then((res: ContactResponse) => this.processContacts(re\
s));
}
```

In the `processContacts` method, call `postMessage` on the `BroadcastChannel` with an `Object` that sets the `type` to `data` and `detail` to the `ContactResponse`, which is expected by `TableComponent` to display columns and rows.

```
processContacts(model: ContactResponse): ContactResponse {
  this.channel.postMessage({
      type: 'data',
      detail: model,
  });
  return model;
}
```

Make a second method that handles when the user changes data in the table.

```
modifyContacts(rows: Array<Contact>) {
  return fetch(this.path, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(rows),
  })
  .then((res) => {
    if (res.status === 200) {
      return res.json();
    } else {
      return {
        columnData: [],
        rowData: [],
      };
    }
  })
  .then((res: ContactResponse) => this.processContacts(re\
s));
}
```

Import the `ContactsService` into `DashboardView.ts` and instantiate the service with the name of the channel (`dashboard-channel`).

```
import { ContactService } from "../../service/contacts";

const contactService = new ContactService("dashboard-chan\
nel");
```

Set a property on the `DashboardView class` named `channelName` to the same string: `dashboard-channel`.

```
export class DashboardView extends HTMLElement {
  channelName: string = 'dashboard-channel';
```

In the `constructor`, call `getContacts` on `contactService`.

```
contactService.getContacts().then((model) => {
  if (model.columnData.length && model.rowData.length) {
    this.onTableData();
  }
});
```

In `onTableData`, listen on the same `BroadcastChannel` for the `change` message and call the `modifyContacts` with the `detail` on that message.

```
onTableData(context) {
  const channel = new BroadcastChannel(this.channelName);
  channel.onmessage = (ev) => {
    if (ev.data.type === "change") {
      contactService.modifyContacts(ev.data.detail);
    }
  }
}
```

Before wrapping up development, let's take care of some loose ends in the application.

AppHeader

In the last chapter, we manually set the location in the address bar in `AppHeader`, although now we have a service that does that.

```
@Listen('click', '.icon')
onLogoClick() {
  document.location.href = '/';
}
```

Import `LocationService` into `AppHeader.ts` and instantiate the service.

```
import { LocationService } from "../../service/location";

const locationService = new LocationService();
```

Replace the line in `onLogoClick` with a call to `locationService.navigate`.

```
@Listen('click', '.icon')
onLogoClick() {
  locationService.navigate('');
}
```

MainView

There's one requirement that had to wait until the user could have an active session after authenticating. On the landing page, when the user has an active session, a link should appear directing them to the dashboard. Import the `SessionService` into `MainView.ts` to add support for checking the user session.

```
import { SESSION, SessionService } from "./../../service/\
session";

const sessionService = new SessionService();
```

In `connectedCallback`, check if the session is `OPEN` and if so, remove the `hidden` attribute from the link on the dashboard. First, add a getter that returns the `HTMLAnchorElement`.

```
get $dashboardLink() {
  return this.shadowRoot.querySelector('.dashboard-link');
}
```

Add the call to `getSesstion` and call `removeAttribute` on the `HTMLAnchorElement`, removing the `hidden` attribute if the user has a session.

```
sessionService.getSession().then((status) => {
  if (status.session === SESSION.OPEN) {
    this.$dashboardLink.removeAttribute("hidden");
  }
});
```

That does it for implementing the features of the application. Let's wrap things up before moving onto the next chapter.

Summary

In this chapter, you built two features of the In application entirely with Web Components: the login and dashboard views. By maintaining a separation of concerns between routing, views, UI components, and services, you were able to quickly scale the

application beyond the initial landing page. Having a library of custom elements at your disposal, along with examples in Storybook enabled you to develop the features fast and efficiently.

In Chapter Ten, you'll take the views you've developed in the last two chapters and refactor them to be server-side renderable, while developing the engine that enables server-side rendering with Node Express and the `@lit-labs/ssr` package.

Server-side Rendering with Declarative Shadow DOM

Chapter Ten

Server-Side Rendering Custom Elements

There are many reasons why you would want to server-side render user interfaces. Server-side rendering UI may increase performance for the end-user because all the content for a HTML document is pre-rendered, reducing the time to paint each element in the browser window. UI components rendered server-side may already be "hydrated" with content. By offloading processing of UI components to a server, the overall time it takes to render HTML can be more predictable. Client-side rendering can be less predictable because different browsers have varied performance.

Server-side rendering promotes SEO compliance, making server-side renderable pages more compatible with search engine crawlers and social media. UI components rendered with a single-page application framework are less compatible with SEO because nothing is rendered in a timely fashion for a web crawler to statically analyze a HTML document. By offloading network requests to the server and hydrating UI components server-side with content, you can improve the security of an application by removing the possibility of exposing sensitive customer data.

For awhile, it was seemingly impossible to server-side render custom elements. Shadow DOM is imperative; you code statements that tell the browser to instantiate a new Shadow Root. This Shadow Root is what provides the ability for custom elements to take advantage of CSS scoping, DOM encapsulation, and composability. While the Shadow DOM specification is powerful, due to its imperative nature, Shadow DOM isn't immediately server-side renderable. Browser specifications aren't natively available in server environments like Node.js. Node.js is built on the V8 JavaScript engine that originated in the Chrome browser, but Node.js lacks browser-specific specifications like the Document Object Model (DOM). If you were going to try to instantiate the `class` that extends `HTMLElement` in a Node.js environment, there would be an error because `HTMLElement` doesn't exist in the context of Node.js, for instance.

The lack of the DOM in Node.js presents an interesting problem when server-side rendering custom elements. All the necessary browser specifications would need to be "shimmed" or "polyfilled" before the custom element `class` could instantiate, then custom logic on the server would need to parse and output a server-side generated template. An npm package that did just that is `@skatejs/ssr`, a package currently in maintenance mode, which was created by Github user @treshugart[81]. `@skatejs/ssr` extended undom[82] and brought support for Custom Elements v1 to Node.js. You could use this package to server-side render custom elements built with Skate[83], or with some configuration, custom elements coded directly with browser specifications. This tool generated templates server-side that enabled sites built with custom elements to be SEO friendly and highly performant. `@skatejs/ssr` worked for static-site generation, but it lacked the ability to "hydrate" components. It became apparent to the web development community a specification that enabled server-side rendering was needed. Enter

[81] https://github.com/treshugart
[82] https://github.com/developit/undom
[83] https://github.com/skatejs/skatejs

Declarative Shadow DOM.

Declarative Shadow DOM

Declarative Shadow DOM is a browser specification that enables Shadow Root to be declaratively instantiated in HTML. Looking at the `AppHeader` component you developed in the last chapter, let's see how Declarative Shadow DOM differs. Instead of imperatively declaring a Shadow Root, you can declare the Shadow Root directly on a HTML template using the `shadowroot` HTML attribute. In the example below that abbreviates the necessary styling and HTML template, `<app-header>` tag name is included in a template. Using Declarative Shadow DOM, the direct child of the `AppHeader` is a HTML template. The HTML template includes a Shadow Root with the HTML attribute `shadowroot` set to the `mode` that would have otherwise been imperatively defined via a call to `attachShadow` on the `AppHeader class`. The template includes the CSS styling and HTML for the component.

```
<app-header>
  <template shadowroot="open">
    <style>
      :host {
        display: flex;
        justify-content: space-between;
        align-items: center;
        height: 100px;
        padding-left: var(--margin-lg);
        padding-right: var(--margin-lg);
        border-bottom: var(--border-default);
      }
    </style>
    <button is="in-button" class="in-button icon logo"></\
button>
    <a href="/login" class="login-link" hidden>Login</a>
```

```
  </template>
</app-header>
```

In the below example, you can compare the Declarative Shadow DOM example above to the imperative style of declaring a Shadow Root you've used to define the template for each component developed in this book.

```
export class AppHeader extends HTMLElement {
  constructor() {
    super();
    const shadowRoot: ShadowRoot = this.attachShadow({ mo\
de: "open" });
    const template = document.createElement("template");
    template.innerHTML = html`
      <style>
        :host {
          display: flex;
          justify-content: space-between;
          align-items: center;
          height: 100px;
          padding-left: var(--margin-lg);
          padding-right: var(--margin-lg);
          border-bottom: var(--border-default);
        }
      </style>
      <button is="in-button" class="in-button icon logo">\
</button>
      <a href="/login" class="login-link" hidden>Login</a>
    `;
    shadowRoot.appendChild(template.content.cloneNode(tru\
e));
  }
}
```

Declarative Shadow DOM enables server-side rendering because

a template is expressed as a `String`. Strings can be manipulated in any JavaScript environment. In the browser, HTML is parsed by the user agent which isn't present in the Node.js environment. If you attempted to use methods that create HTML like `document.createElement` on the server, that method would fail because it doesn't exist. By expressing the template as a `String`, you can pass the template in the HTTP response from the server to the client: a browser that can parse the HTML template as DOM.

Once the browser receives a HTML response with elements that use Declarative Shadow DOM, the user agent parses the HTML template with the `shadowroot` attribute much like it would when calling `attachShadow`. Components that have registered their ES2015 `class` with the `CustomElementRegistry` and rendered with Declarative Shadow DOM don't have to call `attachShadow` any longer in the `class constructor`. This presents a problem for components that imperatively declare their Shadow Root, and components that need to "hydrate" their template during instantiation. If a component imperatively calls `attachShadow` that also has its template defined declaratively, the component will render duplicates of the same template in the browser window.

To prevent this phenomenon, you can check if a Declarative Shadow Root exists prior to imperatively declaring a template for each component. This is possible because any custom element using Declarative Shadow DOM is already upgraded and will already have attached the Shadow Root prior to the `constructor` being called.

```
export class AppHeader extends HTMLElement {
  constructor() {
    super();
    if (!this.shadowRoot) {
      const shadowRoot: ShadowRoot = this.attachShadow({ \
mode: "open" });
      const template = document.createElement("template");
      ...
    }
  }
}
```

When developing custom elements that can be rendered with Declarative Shadow DOM, it's important to separate the logic for template instantiation and component hydration. This allows for easier maintenance of custom elements and makes their instantiation more predictable.

```
export class AppHeader extends HTMLElement {
  constructor() {
    super();
    if (!this.shadowRoot) {
    // instantiate template here
      const shadowRoot: ShadowRoot = this.attachShadow({ \
mode: "open" });
      const template = document.createElement("template");
      ...
    }
    // hydrate the template here
    const linkElement = this.shadowRoot.querySelector('.l\
ogin-link');
    linkElement.textContent = 'Login';
    linkElement.addEventListener('click', () => {
       document.location.href = '/';
    }),
```

```
18    }
19  }
```

@lit-labs/ssr

@lit-labs/ssr[84] is an npm package distributed by Lit at Google for server-side rendering Lit templates and components. While developing with Storybook, you used Lit templates when writing each story. Storybook is dependent on story templates being wrapped in the Lit `html function`. Throughout the course of this book, we opted to code custom elements using just browser specifications and didn't adopt a library like Lit to develop UI components. Had we done so, those `LitElement` would be directly compatible with `@lit-labs/ssr` and benefit from everything that package has to offer. Lit is built on top of web standards, much like the `@in/common` micro-library you coded, although Lit is way more powerful, at the cost of performance. By developing with web standards, we helped ensure the UI components perform as close to the custom element spec itself, which is as performant as you can get in the browser, short of ahead-of-time compilation[85]. The benefits of using a library like Lit can outweigh those performance concerns. The reactive properties and declarative templates alone make Lit an alluring choice for developing UI components.

Using `@lit-labs/ssr` in Node.js, we can opt to use Lit only in the context of declaring the server-side renderable template for each component, keeping our existing UI component library intact. `@lit-labs/ssr` works with vanilla custom elements.

[84] https://www.npmjs.com/package/@lit-labs/ssr
[85] https://github.com/kaleidawave/prism

Challenges

In this chapter, you'll refactor each component from Chapters Eight and Nine to provide Declarative Shadow DOM templates that will be server-side rendered with `@lit-labs/ssr`. You'll use ES2015 template strings to define Declarative Shadow DOM templates in a composable fashion so you can reuse the same template in both imperative and declarative contexts. To help with composability, you'll use another npm package named `es6-template-strings` that resolves interpolated template strings. `es6-template-strings` allows any HTML response that includes Declarative Shadow DOM templates to resolve as `String`, as opposed to `function` references embedded in an ES2015 template string.

```
import resolve from "es6-template-strings";

export const template = () => `
<app-header>
 <template shadowroot="open">
   <style>
   ${resolve(styles)}
   </style>
 ${resolve(shadowTemplate)}
 </template>
</app-header>
`;
```

You'll import the same JavaScript used to declare each view in the application into the context of Node.js, where `@lit-labs/ssr` renders the template. When the web browser requests the HTML document, Node.js will respond with a server-side rendered template.

`@lit-labs/ssr` does a decent job at shimming the browser specifications needed for autonomous custom elements to work in Node.js, although the `class` names customized built-in elements

extend from are not there by default. You'll use tooling provided by `@lit-labs/ssr` to shim any missing specifications needed to render some of the customized built-in elements exported from `@in/ui`.

At the time of writing this, Declarative Shadow DOM isn't available in every browser. Google Chrome and Microsoft Edge have support for Declarative Shadow DOM. Browsers that lack the specification require a "ponyfill" that parses Shadow Roots in a similar way.

By moving to server-side rendering, you'll no longer need the client-side router developed in Chapter Nine. Each route of the application will be handled by the server instead of the SPA in Chapter 10. You will be able to reuse the configuration for the router server-side in Node.js. In this chapter, you'll provide the route configuration to map a specific route to the view component. This view component will when be server-side rendered by `@lit-labs/ssr` and returned in the HTTP response.

After you've declared Declarative Shadow DOM templates, deprecated the client-side router, and shimmed DOM specifications in Node.js, you'll develop Express middleware that server-side renders each template with `@lit-labs/ssr`. If you've never coded in Node.js, have no worries. You'll develop the Express server with JavaScript to reduce some complexity from compiling TypeScript, while completing tasks in this chapter using step-by-step instructions.

What you will build

- Declarative Shadow DOM templates for server-side rendering
- Express middleware that server-side renders custom element templates with `@lit-labs/ssr`
- Router schema to support client and server implementations

What you will learn

- What is Declarative Shadow DOM?
- How to server-side render web components with `@lit-labs/ssr` and Express
- How to shim Web Component specifications in Node.js
- How to reuse client custom element templates server-side
- Best Practices for server-side rendering that impact performance and SEO

Acceptance criteria

- Each route is server-side rendered

Getting started

Navigate to the project directory:

```
cd fullstack-web-components
```

Checkout a new branch named chapter-10 from where you left off at the end of chapter 9.

```
git checkout -b chapter-10
```

Declarative Shadow DOM Templates

Declarative Shadow DOM templates aren't much different from the HTML templates you've been coding with all along. The same rules

apply. Global CSS styling still can't cross the Shadow boundary. If a customized built-in element is found in the template, you'll need to provide the styles for that component in the template as well. The imperative logic you coded in previous chapters that includes the appropriate `<style>` tag via the `@Component` decorator won't be available server-side, where these Declarative Shadow DOM templates will be used to server-side render the template of each route in the application.

The only real difference is that any custom element declared in a Declarative Shadow DOM template is represented by its tag name. When imperatively creating templates, you didn't need to encapsulate the template with the tag name of the custom element. That was assumed.

To provide Declarative Shadow DOM templates for each of the components in the app, you'll export a new `const` named `template` in each of the component files you coded in Chapters Eight and Nine. This `const` will equal a `function` that returns a template string. This format has two main advantages: a data model could be passed through the `function` arguments and passed to the template, hydrating the template with content on the server. Secondly, we can reuse existing `const` for CSS styles and HTML template that were originally passed to the `@Component` decorator.

```
export const template = (data?: { label: string }) => `
<footer>
  <template shadowroot="open">
     <style>
     ${styles}
     </style>
     <label>${data.label}</label>
  </template>
</footer>
`;
```

This approach will only get you so far. Unfortunataly, the above

example won't fully resolve the interpolated strings (`${styles}` or `${data.label}`) found in the template string. For that, we'll need to use a second package named `es6-template-strings`. The `resolve function` exported from the es6-template-strings will print the content of each interpolated `string`.

```
import resolve from "es6-template-strings";
```

After importing `resolve` we can ensure each interpolated `string` prints the necessary content server-side.

```
export const template = (data?: any) => `
<cookie-footer>
  <template shadowroot="open">
     <style>
     ${resolve(styles)}
     </style>
     ${resolve(shadowTemplate)}
  </template>
</cookie-footer>
`;
```

Several of the components you'll server-side render have a child `ButtonComponent` customized built-in element. If the styling for that custom element will also be server-side rendered, you'll need to find a way to import the styling from the `@in/ui` package.

```
import { buttonStyles } from "@in/ui";
```

Open `packages/component/src/button/Button.ts` and export the `buttonStyles` from the file.

```
export const buttonStyles = css`
```

Follow up by exporting the same `buttonStyles` from `packages/component/src/button/index.ts`.

```
export { buttonStyles, ButtonComponent } from "./Button";
```

Build the `@in/ui` package before proceeding so you can import `buttonStyles` into other files.

```
lerna exec --scope @in/ui -- yarn build
```

If you have already started the application server, stop and restart the development server.

```
yarn start:app
```

CookieFooter

We're going to start refactoring each component from the ground up. That means beginning with `CookieFooter` and `AppHeader` before moving onto `MainView`, `LoginView`, and `DashboardView`. We don't have to be so concerned with `Background`, which lazyloads a rasterized image and can't be easily server-side rendered. We can however server-side render all vector-based SVG elements because those are declared like any other DOM element.

Begin in `packages/client/src/component/footer/CookieFooter.ts` by importing `buttonStyles` from `@in/ui` and `resolve` from `es6-template-stings`.

```
import { buttonStyles } from "@in/ui";
import resolve from "es6-template-strings";
```

At the bottom of the file, declare and export a new `const` named `template`, making it equal to a `function` that returns a template string. This will be the format for all server-side rendered templates. In the template string, wrap the entire template in a `<cookie-footer>` tag. Declare a new HTML template as the direct child of `CookieFooter`, adding the `shadowroot` attribute and making it equal to `open` to establish this template uses the Declarative Shadow DOM spec.

Add a `<style>` tag and using string interpolation with `resolve`, pass the `styles` used by `CookieFooter` and the `buttonStyles` into the `<style>` tag. This will eventually have the effect of printing the entire CSS styling needed for `CookieFooter` to display properly. Next, after the `<style>` tag, use string interpolation and `resolve` again to provide the template of `CookieFooter` stored in `shadowTemplate`.

```
export const template = () => `
<cookie-footer>
  <template shadowroot="open">
     <style>
     ${resolve(styles)}
     ${resolve(buttonStyles)}
     </style>
     ${resolve(shadowTemplate)}
  </template>
</cookie-footer>
`;
```

At this point, we can't test server-side rendering of this or any template developed in this section, but you could log the output of `template()` and observe how the template would be rendered server-side in Chrome Dev Tools.

```
console.log(template());
```

AppHeader

Repeat the same steps in `packages/client/src/component/header/Header.ts`. Import `buttonStyles` and `resolve`.

```
import { buttonStyles } from "@in/ui";
import resolve from "es6-template-strings";
```

Export a new `const` named `template` that returns the template for the `AppHeader`, being sure to declare the `<app-header>` tag, a HTML template that uses Declarative Shadow DOM. Provide the styling and template needed for the component to display.

```
export const template = () => `
<app-header>
 <template shadowroot="open">
   <style>
   ${resolve(styles)}
   ${resolve(buttonStyles)}
   </style>
 ${resolve(shadowTemplate)}
 </template>
</app-header>
`;
```

Now both of the reusable components have server-side renderable templates. You'll reference those templates in the context of the view layer. You'll use the `template` exported from `Header.ts` and `CookieFooter.ts` to compose the layout in a Declarative Shadow DOM HTML template.

MainView

We'll need to address the same concerns in the`MainView` template found in `packages/client/src/view/main/MainView.ts`. `MainView` takes care of its own template's styling, but to style the `AppHeader` and `CookieFooter`, you'll need to reuse the templates from those two component implementations. Just like in `AppHeader.ts` and `CookieFooter.ts`, import `resolve` into `MainView.ts` from the `es6-template-strings` package.

```
1  import resolve from "es6-template-strings";
```

Another problem we need to take care of in the view layer is how to provide all the code for the components in each view. Right now all the components are bundled in one JavaScript file in `packages/client/src/index.ts` but that should change if we want to optimize each route, only bundling the components needed for that view. To help with the transition to multiple bundles that are server-side rendered, import all the necessary dependencies of `MainView` in `MainView.ts` (`AppHeader`, `CookieFooter`, and `Background`). Import the `template` exported from `Header.ts` and `CookieFooter.ts`, but since they are named the same, use `as` to rename the exports `HeaderTemplate` and `FooterTemplate`.

```
1  import {
2    AppHeader,
3    template as HeaderTemplate,
4  } from "../../component/header/Header";
5  import {
6    CookieFooter,
7    template as FooterTemplate,
8  } from "../../component/footer/CookieFooter";
9  import { Background } from "../../component/background/Ba\
10 ckground";
```

There's a precarious problem to solve with the `MainView` template. We want to reuse the `HeaderTemplate` and `FooterTemplate` functions for server-side rendering but those elements are referenced in `shadowTemplate`. The content in between the header and footer could be reused between the client-side and server-side template as well, so extract the two marketing sections into another template string stored in a `const` named `contentTemplate`.

```
const contentTemplate = html`
  <div
    is="in-bg"
    background="/style/asset/timon-studler-BIk2ANMmNz4-un\
splash.jpg"
  >
    <div class="blurb half right">
      <h2>Your Last Contact List</h2>
      <p>
        Lorem ipsum dolor sit amet, consectetur adipiscin\
g elit. Quod eo
        liquidius faciet, si perspexerit rerum inter eas \
verborumne sit
        controversia.
      </p>
      <a href="/dashboard" class="cta dashboard-link" hid\
den>View Contacts</a>
    </div>
  </div>
  <div
    is="in-bg"
    background="/style/asset/valiant-made-zBkVp3E2CnE-uns\
plash.jpg"
    class="light"
  >
    <div class="blurb">
      <h2>Turn Group Chats Into Live Events</h2>
```

```
      <p>
        Indicant pueri, in quibus ut in speculis natura c\
ernitur. Itaque ad
        tempus ad Pisonem omnes. Quis hoc dicit? Philosop\
hi autem in suis
        lectulis plerumque moriuntur. Videmus in quodam v\
olucrium genere non
        nulla indicia pietatis, cognitionem, memoriam, in\
 multis etiam desideria
        videmus. Quid sequatur, quid repugnet, vident. At\
qui pugnantibus et
        contrariis studiis consiliisque semper utens nihi\
l quieti videre, nihil
        tranquilli potest.
      </p>
    </div>
  </div>
`;
```

Reference `contentTemplate` in the necessary part of `shadowTemplate`, the content area of the `<div id="content-root">`.

```
const shadowTemplate = `
  <app-header></app-header>
  <div id="content-root">${contentTemplate}</div>
  <cookie-footer hidden></cookie-footer>
`;
```

Near the end of `MainView.ts`, export a `const` named `template` that is a `function` where the Declarative Shadow DOM HTML template is returned. Inject the `styles` of `MainView` into the template, along with `HeaderTemplate`, `contentTemplate`, and `FooterTemplate` using `resolve`.

```
export const template = () => `
<main-view>
  <template shadowroot="open">
    <style>
     ${resolve(styles)}
    </style>
    ${resolve(HeaderTemplate())}
    <div id="content-root">
      ${contentTemplate}
    </div>
    ${resolve(FooterTemplate())}
  </template>
</main-view>
`;
```

Export all the necessary components that should be bundled with this view, including children of `AppHeader` and `CookieFooter` like `ButtonComponent`.

```
export { ButtonComponent } from "@in/ui";
export { AppHeader, CookieFooter, Background };
```

LoginView

Repeat the same steps for `LoginView`. Import `resolve` and the necessary dependencies from `Header.ts`.

```
import resolve from "es6-template-strings";
import {
  AppHeader,
  template as HeaderTemplate,
} from "../../component/header/Header";
```

Extract the content from the `shadowTemplate` in a new `const` named `contentTemplate` that can be reused on the client and server.

```
const contentTemplate = html`
<in-card style="max-width: 320px">
<h4 slot="header">Login</h4>
<form
  name="foo"
  slot="content"
>
  <fieldset>
      <legend>Login Form</legend>
      <label for="username">Username</label>
      <in-textinput
        type="text"
        id="username"
        name="username"
        required
        minlength="5"
        class="form-control"
      ></in-textinput>
      <label for="password">Password</label>
      <in-textinput
        type="password"
        id="password"
        name="password"
        required
        minlength="5"
        class="form-control"
      ></in-textinput>
      <button class="primary submit form-button" is="in-b\
utton">Submit</button>
</form>
</in-card>
`;
```

Inject the new `contentTemplate` into the appropriate part of `shadowTemplate`.

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root">${contentTemplate}</div>
`;
```

Export a new `template` that injects `styles`, `HeaderTemplate`, and `contentTemplate` with `resolve`.

```
export const template = () => `
<login-view>
  <template shadowroot="open">
    <style>
     ${resolve(styles)}
    </style>
    ${resolve(HeaderTemplate())}
    <div id="content-root">
     ${contentTemplate}
    </div>
  </template>
</login-view>
`;
```

Ensure all components needed for this view are exported from the bundle.

```
export { ButtonComponent, CardComponent, TextInputCompone\
nt } from "@in/ui";
export { AppHeader };
```

DashboardView

Repeat the same steps in `DashboardView.ts`. Import any needed dependencies at the top of the file.

```
import resolve from "es6-template-strings";
import {
  AppHeader,
  template as HeaderTemplate,
} from "../../component/header/Header";
```

Extract the content into a separate template that can be referenced in the Declarative Shadow DOM HTML template. Notice how we're not trying to server-side render the entire table in this example, but that could be possible with some additional effort.

```
const contentTemplate = html`
  <h1>Contacts</h1>
  <in-tablecard channel="dashboard-channel"></in-tablecar\
d>
`;
```

Inject the `contentTemplate` into the existing `shadowTemplate`.

```
const shadowTemplate = html`
  <app-header></app-header>
  <div id="content-root">${contentTemplate}</div>
`;
```

Export a `template` that returns the Declarative Shadow DOM template for `DashboardView`.

```
export const template = () => `
<dashboard-view>
  <template shadowroot="open">
    <style>
     ${resolve(styles)}
    </style>
    ${resolve(HeaderTemplate())}
    <div id="content-root">
      ${contentTemplate}
    </div>
  </template>
</dashboard-view>
`;
```

Export all the necessary components to display `DashboardView`.

```
export {
  ButtonComponent,
  CardComponent,
  TableComponent,
  TableCardComponent,
  TrComponent,
  TdComponent,
  TextInputComponent,
} from "@in/ui";
export { AppHeader };
```

That's it for refactoring each application custom element to be server-side renderable. We didn't touch every component in the `@in/ui` library, although those components that use Shadow DOM could also expose a Declarative Shadow DOM template. Customized built-in elements just need to export the styling for the component so the CSS styles can be server-side rendered.

Webpack

To complete the setup needed for each route to be bundled separately for each server-side rendered route, modify the `webpack.config.js` at `packages/client/webpack.config.js` to bundle additional entry points, one for each view.

```
main: './src/view/main/MainView.ts',
login: './src/view/login/LoginView.ts',
dashboard: './src/view/dashboard/DashboardView.ts',
```

Routing

For server-side rendering, we no longer need the entry point of the client application at `packages/client/src/index.ts` to export components. The custom elements that make up each view are available in each bundle built by Webpack. The responsibility of routing will also shift from client to server, so we can deprecate the client-side router, but opt to keep the same route configuration stored in `routes` to configure routing server-side instead. Modify `packages/client/src/index.ts` so `index.ts` only exports `routes`.

```
export { routes } from "./routes";
```

In this section, you exported Declarative Shadow DOM HTML templates that will be imported into the context of Node.js later in this chapter. These changes have the undesireable effect of taking the client offline. Custom elements will no longer appear for each route in the browser until you complete the Express middleware that renders each route. Before you even tackle that challenge, you'll need to provide a shim for custom element specificiations that are not available in Node.js.

Shim

In the last section, you exported three new JavaScript bundles corresponding to each route of the application (/, /login, and /dashboard). If we're going to import client-side bundles in Node.js to get their exported Declarative Shadow DOM HTML templates, Node.js will attempt to parse the JavaScript in the bundle. Declarative Shadow DOM, along with all custom element specifications, is still a browser-based technology, no way around it. Node.js runs on a server that lacks browser specific DOM specifications. You'll need to provide mocked implementations of the custom elements v1 specification so the components you've coded in this book can be interpreted by the Node.js runtime.

`@lit-labs/ssr`, the package you'll use to server-side render the Declarative Shadow DOM HTML templates exports a `function` named `installWindowOnGlobal`. When executed in Node.js, `installWindowOnGlobal` provides coverage for autonomous custom elements portion of the Custom Elements v1 specification, but lacks support for the customized built-ins several components from the `@in/ui` package and `Background` in the client package rely on. If the application just included autonomous custom elements, we could import `installWindowOnGlobal`, call the `function` and have complete coverage for running custom element code in Node.js.

The shim is stored in a separate package in the repository named `@in/dom-shim` at `packages/shim`. In this section, you'll code the shim, build the `@in/dom-shim` package, so that in the next section you can import the shim and use it.

In `packages/shim/index.ts` import the `installWindowOnGlobal` method from `@lit-labs/ssr`.

```
1 import { installWindowOnGlobal } from "@lit-labs/ssr/lib/\
2 dom-shim.js";
```

Since you need to support customized built-ins as well, you'll need to mock portions of custom elements specification. This includes `HTMLElement` and several of its `class` methods, including `attributes`, `setAttribute`, `removeAttribute`, `hasAttribute`, `attachShadow`, and `getAttribute`. Using code from the `@lit-labs/ssr` package, mock the `HTMLElement class` so we can run custom elements in the context of Node.js. Starting with the `attributes` of a `HTMLElement`, create a `new WeakMap()` and reusable `function` named `attributesForElement` that will help mock each of the necessary methods on the `HTMLElement class`.

```
const attributes = new WeakMap();
const attributesForElement = (element) => {
  let attrs = attributes.get(element);
  if (!attrs) {
    attributes.set(element, (attrs = new Map()));
  }
  return attrs;
};
```

Declare a new `class` named `Element`, leaving it empty. `HTMLElement` actually extends from the `Element class` in the browser, so you'll mock that same behavior here, only that `Element` doesn't exist in Node.js. Define the `HTMLElement class` and establish the first getter named `attributes`, which calls the `attributesForElement function` you defined earlier and returns an `Array` of attribute names and values.

```
class Element {}
class HTMLElement extends Element {
  get attributes() {
    return Array.from(attributesForElement(this)).map(([n\
ame, value]) => ({
      name,
      value,
    }));
  }
}
```

Define the remaining methods on `HTMLElement` that need to be mocked for `HTMLElement class` to instantiate properly in Node.js.

```
  setAttribute(name, value) {
    attributesForElement(this).set(name, value);
  }
  removeAttribute(name) {
    attributesForElement(this).delete(name);
  }
  hasAttribute(name) {
    return attributesForElement(this).has(name);
  }
  attachShadow() {
    return { host: this };
  }
  getAttribute(name) {
    const value = attributesForElement(this).get(name);
    return value === undefined ? null : value;
  }
```

Now that `HTMLElement` is mocked, define `class` for each of the customized built-in elements that extend from a `class` available in the browser, but not Node.js. For example, `ButtonComponent` extends `HTMLButtonElement`, but `HTMLButtonElement` is `undefined` in Node.js.

Declare `HTMLButtonElement` here, along with `HTMLTableElement`, `HTMLTableCellElement`, `HTMLTableRowElement` and `HTMLDivElement`.

```
class HTMLButtonElement extends HTMLElement {}
class HTMLTableElement extends HTMLElement {}
class HTMLTableCellElement extends HTMLElement {}
class HTMLTableRowElement extends HTMLElement {}
class HTMLDivElement extends HTMLElement {}
```

Internally, `installWindowOnGlobal` mocks `HTMLElement` for us, but doesn't expose that `class` as an `export` for you to extend other `class` from it. That's why we needed to duplicate the `HTMLElement` mock in `index.ts`. `installWindowOnGlobal` does allow us to add mocks when the shim executes, by calling the `function` with an `Object` where each of the `class` is stored as a property on that `Object`. So we can execute `installWindowOnGlobal` in the context of the Express application, wrap the call to `installWindowOnGlobal` in another `function` named `installShimOnGlobal` and export that `function`. Call the `installShimOnGlobal function` so it executes very early in the Node.js environment.

```
const installShimOnGlobal = () => {
  installWindowOnGlobal({
    HTMLButtonElement,
    HTMLTableElement,
    HTMLTableCellElement,
    HTMLTableRowElement,
    HTMLDivElement,
  });
};
installShimOnGlobal();
```

```
export { installShimOnGlobal };
```

In the next section, while developing the Express middleware that server-side renders custom elements, you'll import the `installShimOnGlobal function` from `@in/dom-shim` and call it prior to other code running, effectively polyfilling portions of the custom elements v1 specification required for browser-based code to run on a server. Go ahead and build the `@in/dom-shim` package so it's possible to import the `function` in the next section.

```
lerna exec --scope @in/dom-shim -- yarn build
```

Ponyfill

In Node.js you needed to polyfill the custom elements v1 specification just enough so the JavaScript exported from client bundles can run in Node.js. Similarly, we need to ensure Declarative Shadow DOM templates are parsed properly by browsers that don't support that specification yet. At the time of writing this, Google Chrome and Microsoft Edge are the only two browsers that natively support Declarative Shadow DOM. To provide support for Declarative Shadow DOM in other browsers, we can't use a polyfill. The entire Document Object Model (DOM) needs to be fully loaded before any code can begin to interpret the HTML document, parsing Declarative Shadow DOM Shadow Roots. A package exists that does just that named `@webcomponents/template-shadowroot`, but in the form of a ponyfill. Ponyfills execute at the end of the `document.body`, while polyfills are typically found in the `document.head`, so code in the polyfill can execute prior to any DOM loading or other JavaScript being parsed. Ponyfills typically execute after JavaScript in the `document.head` and the `document.body` has loaded. While developing the HTML template that is server-side rendered, you'll

need to inject the polyfill at the end of the `document.body` before the template is sent to the browser client.

Store the ponyfill for now in `packages/client/src/ponyfill.ts`. In the next section, you'll find a way to import this code into the server-side rendered template.

```
import {
  hasNativeDeclarativeShadowRoots,
  hydrateShadowRoots,
} from "@webcomponents/template-shadowroot/template-shado\
wroot.js";
if (!hasNativeDeclarativeShadowRoots) {
  hydrateShadowRoots(document.body);
}
```

Now that you've shimmed the custom elements v1 specification in Node.js and provided a ponyfill to Declarative Shadow DOM for browsers that don't support the specification yet, move onto developing the Express middleware responsible for server-side rendering custom element templates.

Express Middleware

Express.js is a popular Node.js framework that handles common use cases for server-side web development like handling http requests, routing, templating, and debugging. This project uses Express.js to provide backend API services for the web application. When the web application made HTTP requests to `/api/login` for instance, Express handled that request, checked the username and password against a database, then sent a response depending on the validity of the username and password. Express uses middleware to achieve this. API middleware defined in the file `packages/server/src/middleware/api.js` handles multiple types

of HTTP requests following REST API design guidelines. That middleware is then imported and injected into the Express application in the entry point (`packages/server/src/index.js`).

```
import apiRouter from './middleware/api.js';
...
const app = express();
...
app.use('/api', apiRouter);
```

In the above example, the instance of `express.Router` defined in `packages/server/src/middleware/api.js` is imported into the entry point of the Express application. The application is instantiated by calling `express()`. Afterward, middleware can be declared for the Express application. Whenever a HTTP request is made at `/api`, the `apiRouter` will handle the request.

Most of the development of the Express server was handled for you. This was done so you could concentrate on coding custom elements, with one notable exception: Express middleware that handles server-side rendering Declarative Shadow DOM HTML templates. To achieve this, you'll develop the Express middleware that handles server-side rendered HTML responses at the following routes:

- /
- /login
- /dashboard

Right now, every request by default loads the `index.html` of the client-side application. This was necessary because in the previous chapter you were dealing with a single-page application.

```
app.get("/*", (req, res) => {
  res.sendFile(clientPath("index.html"));
});
```

Later in this chapter, you'll deprecate that handling of the SPA. After importing the `ssr` middleware, you'll map the HTTP requests for the three routes to the middleware.

```
import ssr from "./middleware/ssr.js";
...
app.get("/", ssr);
app.get("/login", ssr);
app.get("/dashboard", ssr);
```

Development in this section is done like any other before it. You may continue by running the command `yarn start:app` which starts both the client and server development environments.

Client

Before working on the Express middleware, there's still a bit of preparation work that needs to be done on the client. You recently deprecated the client-side SPA router, but left the `routes` around to use them in the context of the server-side renderer. It wasn't made apparant at the time, but this change was necessary to expose the Declarative Shadow DOM HTML templates somewhere that mapped each template to a given route.

The `Route` configuration already defines the route of each view, but you'll need to add the template for each `Route` to the configuration, so later you can use this configuration in the Express middleware. In `packages/client/src/routes.ts`, import all three of the templates from their respective files.

```
import { template as dashboardTemplate } from "./view/das\
hboard/DashboardView";
import { template as loginTemplate } from "./view/login/L\
oginView";
import { template as mainTemplate } from "./view/main/Mai\
nView";
```

Add a new property to each `Route` called `template`, specifying the HTML template for each view to the route it should map to. For instance, `mainTemplate` is the Declarate Shadow DOM HTML template that should render at `/`, while `dashboardTemplate` should display when the user navigates to `/dashboard`.

```
export const routes = [
  {
    path: "/",
    component: "main",
    tag: "main-view",
    title: "In",
    template: mainTemplate,
  },
  {
    path: "/login",
    component: "login",
    tag: "login-view",
    title: "Login",
    template: loginTemplate,
  },
  {
    path: "/dashboard",
    component: "dashboard",
    tag: "dashboard-view",
    title: "Contact Dashboard",
    template: dashboardTemplate,
  },
];
```

With that configuration complete on the client, it's time to code the Express middleware.

Server

To get started coding the Express middleware that will server-side render each view, make a new file named `ssr.js` in the `packages/server/src/middleware` directory.

```
touch packages/server/src/middleware/ssr.js
```

Import the `routes` you just exported from the client application using a relative path for now. This could very well be done using the package name `@in/client`, assuming that package is built.

```
import { routes } from "./../../../client/dist/index.js";
```

In this first part of the `ssr.js`, you'll have to wrangle paths to different files. You're going to write some code that reads the content of several .html, .css, and .js files and then prints the content of those files to a server-side rendered template. To deal with the paths to those files in the local filesystem, import `resolve` from path.

```
import { resolve } from "path";
```

Import `readFileSync` from fs, which will be used to read the content of a file at a given path on the local filesystem.

```
import { readFileSync } from "fs";
```

If you haven't already, build the project before proceeding to ensure all packages are built property and their distributed files can be read by the Express server.

```
yarn build
```

Read Client Bundles

Most of the files that should be returned in the response of the HTML document are found in one of two locations: `@in/client` and `@in/style` package's `dist` directories. Code two reusable `functions` that return the path of a file from either of these two directories. Inject the filename into the first argument of the `function`. Use `resolve` to return the path of the file, relative to where the script is run. `process.cwd()` returns the path where the server is run (`packages/server`). Use a relative path to find the dist directories of the `@in/client` and `@in/style` packages, passing the `filename` argument to the end of the path, so the `function` returns a path to the filename.

```
const clientPath = (filename) =>
  resolve(`${process.cwd()}../../client/dist/${filename}`\
);
const stylePath = (filename) =>
  resolve(`${process.cwd()}../../style/dist/${filename}`);
```

Declare two similar `function` that return the content of the file at the path returned by either `clientPath` or `stylePath`. `readFileSync` will read the content of the file synchronously. By calling `toString` you can easily convert the file contents to a `string`.

```
const readClientFile = (filename) =>
  readFileSync(clientPath(filename)).toString();
const readStyleFile = (filename) =>
  readFileSync(stylePath(filename)).toString();
```

In the next part you'll read the content of files in the `@in/client` and `@in/style` packages, then minify the content for performance reasons, before inserting the content into the HTML document.

Minify CSS and HTML

Before handling the Declarative Shadow DOM templates, we need to figure out a way to render the global CSS styling that contains CSS variables used in every component and the HTML templates that can be referenced by any component. The CSS styles are available in the file at `packages/style/dist/style.css` and the HTML templates are in `packages/client/dist/template.html`. You just coded two `functions` that can read the content of files in these directories, although there's still some processing of these files that needs to happen. Minifying the content of `style.css` and `template.html` will help reduce the size of the overall HTML document, making it faster for browsers to request the document. To minify the content of both files, import `minify` from the `html-minifier-terser` package.

```
import { minify } from "html-minifier-terser";
```

Make a new `const` named `styles`. Using top-level `await`, call `minify`, passing in the file content returned by `readStyleFile("style.css")`. Configure `minify` so that CSS is optimized, all comments are removed, and whitespace is removed, ensuring the most optimal global CSS possible.

```
const styles = await minify(readStyleFile("style.css"), {
  minifyCSS: true,
  removeComments: true,
  collapseWhitespace: true,
});
```

Do the same for the HTML templates stored in `template.html`.

```
const htmlTemplates = await minify(readClientFile("templa\
te.html"), {
  minifyCSS: true,
  removeComments: true,
  collapseWhitespace: true,
});
```

Both `styles` and `htmlTemplates` return `string` you'll be able to inject directly into the HTML document which will be sent in the initial response from the server.

SSR

The `@lit-labs/ssr` package provides tools for rendering Lit templates that use the `html` function exported from the lit package. The Declarative Shadow DOM HTML templates exported in each of the application components don't use Lit, instead preferring to export `string` including HTML templates coded entirely with browser specifications. Because of that architectural design, you'll need to wrap each template in a call to `unsafeHTML` which tells Lit to render HTML instead of just a `string`. The `render` function exported from `@lit-labs/ssr` accepts a `LitTemplate`. Import both `render` and `html` into `ssr.js`.

```
import { render } from "@lit-labs/ssr/lib/render-with-glo\
bal-dom-shim.js";
import { unsafeHTML } from "lit-html/directives/unsafe-ht\
ml.js";
import { html } from "lit";
```

Make a new `function` named `sanitizeTemplate` that accepts one argument: the `template` exported from each view bundle. `sanitizeTemplate` returns an asynchronous `function` that calls `html` with a template string, where the `template` is processed by `unsafeHTML`.

```
export const sanitizeTemplate = async (template) => {
  return html`${unsafeHTML(template)}`;
};
```

Export the middleware from the file with `export default`. All Express middleware are asynchronous, formatted as a `function` with three arguments: the request and response represented by `Object`, and next (denoting the "next" middleware to be called).

```
export default async (req, res, next) => {};
```

The first thing you'll need to do in this `function` is use the current request URL found at `req.url` to match the request against an available route in the configuration. To parse the URL at `req.url` you'll need another tool named `url` imported from url.

```
import url from "url";
```

Loop over the `routes Array`, using `find` to return a match where the route's `path` equals the `pathname` returned by `url.parse`.

```
const route = routes.find((rt) => rt.path === url.parse(r\
eq.url).pathname);
```

If the `route` is not found with `find`, `route` will be `undefined`. Detect this use case and using `res.redirect`, rediect the request to a special route at `/404` that handles requests that do not exist.

```
if (route == undefined) {
  res.redirect(301, "/404");
  return;
}
```

Finally, create a new `const` named `template`. Calling `sanitizeTemplate` with `await`, pass in the template string returned by `route.template()`, the template exported by the bundle at that route.

```
const template = await sanitizeTemplate(route.template());
```

Now that you have the template that should appaer in the `document.body`, build out the rest of the document using a JavaScript Generator. This is the preferred style for generating a server-side rendered document with `@lit-labs/ssr`, so we're not only sticking to convention by using a generator, but also coding with an extremely powerful and performant way to iterate over the template and produce an HTML document.

Declare a new Generator named `renderApp`, denoted by the `*` after the `function` keyword. `renderApp` takes two arguments, the `route Object` and the `template` returned from `sanitizedTemplate`. Using `yield` statements, build out the necessary parts of the HTML document, being sure to call `render` in between a containing `HTMLDivElement` with `id` attribute set to `root`, like you had done for the client.

```
function* renderApp(route, template) {
  yield `
    <!doctype html>
    <html>
      <head></head>
      <body>
      <div id="root">`;
  yield* render(template);
  yield `</div>
    </body>
    </html>`;
}
```

Declare a new `const` named `ssrResult` and make it equal to what the generator returns, passing in the `route` and `template` to the `renderApp function`.

```
const template = await sanitizeTemplate(route.template());
const ssrResult = await renderApp(route, template);
```

The generator will return a stream of data, so you need another function that handles that stream and concatenates the output of the stream to a `string`. Declare a new `function` named `streamToString` that does exactly that. Import `Buffer` to handle the stream.

```
import { Buffer } from "buffer";
```

```
async function streamToString(stream) {
  const chunks = [];
  for await (let chunk of stream) {
    chunks.push(Buffer.from(chunk));
  }
  return Buffer.concat(chunks).toString("utf-8");
}
```

Import `Readable` for the next part so you can read the output of `renderApp`.

```
import { Readable } from "stream";
```

Make another `function` named `renderStream` that takes the `stream` as an argument and returns the synchronous `streamToString`. Use `Readable` to read the content of the `stream` as a `Buffer`.

```
async function renderStream(stream) {
  return await streamToString(Readable.from(stream));
}
```

Add a new `const` named `stream`, passing in the `ssrResult` to `renderStream`.

```
const template = await sanitizeTemplate(route.template());
const ssrResult = await renderApp(route, template);
let stream = await renderStream(ssrResult);
```

Handle the response, sending a positive "200 Success" code, along with the contents of the `stream`.

```
res.status(200).send(stream);
```

At this point, any route that uses this middleware should return a proper HTML document with a server-side rendered view.

Entry Point

To attach the middleware to specific routes, import `ssr` into `packages/server/src/index.js`.

```
import ssr from "./middleware/ssr.js";
```

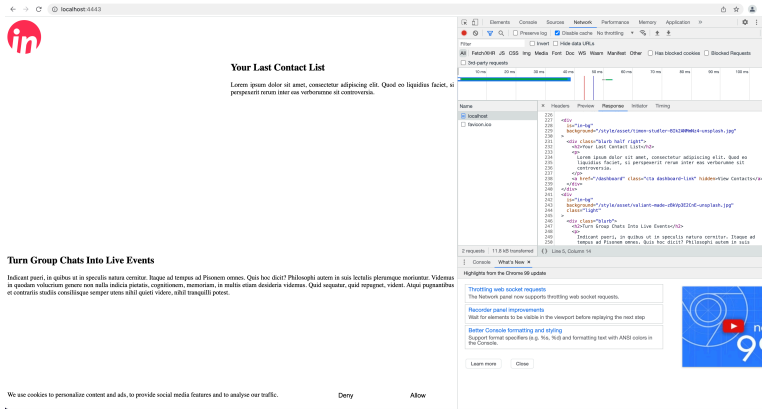
Deprecate the existing middleware that returns the client-side SPA either by deleting or commenting out the following lines.

```
// app.get('/*', (req, res) => {
//   res.sendFile(clientPath('index.html'));
// });
```

Add handlers for each route that map requests to `/`, `/login`, and `/dashboard` to the `ssr` middleware.

```
app.get("/", ssr);
app.get("/login", ssr);
app.get("/dashboard", ssr);
```

After the development environment refreshes, you should observe the following in the browser window at http://localhost:4443/.



The output of the middleware is working. You can tell by inspecting the network response and observing that the template for each route is server-side rendered. You're still missing some styling and client-side logic, but you've done it! Let's fill in some of the missing pieces before finishing the application.

Filling In Gaps

If you inspect the document closely, it appears `@lit-labs/ssr` is printing empty Declarative Shadow DOM HTML templates for some reason. In `ssr.js`, remove empty templates by postprocessing the stream with a `RegExp` that finds all instances of empty templates and replaces them with empty `string`.

```
stream = stream.replace(/<template shadowroot="open"><\/t\
emplate>/g, "");
res.status(200).send(stream);
```

Turning your attention to the `document.head`, provide the title for the document in the `<title>` tag, sourcing the title from the `route`.

Build tooling deploys the fonts with the server in a directory that is accessible by browser requests at `/style/font/`. Use a `<style>` tag to inject the fonts for the application with `@import` statements that point to the necessary files. Finally, inject the server-side optimized global `styles` in another `<style>` tag. This should provide more styling in the view.

```
function* renderApp(route, template) {
  yield `
    <!doctype html>
    <html>
      <head>
          <title>${route.title}</title>
          <style rel="stylesheet" type="text/css">
              @import url("/style/font/Lato-Regular.ttf");
              @import url("/style/font/Lato-Black.ttf");
          </style>
          <style rel="stylesheet" type="text/css">${style\
s}</style>
      </head>
```

Still in the `document.head`, provide the ncessary polyfills for browsers that don't support all of the Web Components specifications, reading the content of `polyfill.js` and injecting the content of the file into a `<script>` tag.

```
<script type="module">${readClientFile("polyfill.js")}</s\
cript>
```

Add the HTML templates referenced by several components after the starting `<body>` tag.

```
    </head>
      <body>
      ${htmlTemplates}
```

Inject the contents of `ponyfill.js` near the closing `</body>` tag. Similarly, inject the bundle for each route by referencing the name of the bundle on `route.component` and reading the file from the client dist directory.

```
  yield* render(template);
  yield `</div>
    <script type="module">${readClientFile(`ponyfill.js`)\
}</script>
    <script type="module">${readClientFile(`${route.compo\
nent}.js`)}</script>
    </body>
    </html>`;
}
```

Congratulations! You've server-side rendered the application using `@lit-labs/ssr`. Let's wrap up development with what you learned in this chapter.

Summary

Declarative Shadow DOM provides the ability to encapsulate custom elements with Shadow DOM in a declarative, template-based approach that allows custom elements to be server-side rendered. In this chapter, you provided Declarative Shadow DOM HTML templates for each of the autonomous custom elements in the application and refactored the client application by deprecating the router. After providing the templates on the `Route` configuration, you shimmed the necessary browser specifications so

client-side custom element implementations could be interpreted in the context of Node.js. You developed Express middleware using `@lit-labs/ssr` that used the same route configuration to server-side render custom element templates with Declarative Shadow DOM.

By finishing the content of Chapter Ten you've completed all the exercises available in Fullstack Web Components.

Fullstack Web Components

Conclusion

Congratulations! You've completed the exercises included in Fullstack Web Components. You coded several UI components using the set of specifications known as Custom Elements v1. You used Shadow DOM to encapsulate a component's HTML template and CSS styling. Upon further investigation of custom elements v1, you found that only autonomous and form-associated custom elements can leverage Shadow DOM. Customized built-in elements lack this capability due to how built-ins extend from common HTML elements, some of which either lack the ability to employ Shadow DOM, or in the case of `HTMLVideoElement`, already utilize Shadow DOM and cannot be overwritten. While this may diminish the use-value of customized built-ins, you developed against this specification with some novel use cases: `ButtonComponent` and `TableComponent` to name a couple. While autonomous custom elements enable the creation of fully custom user interfaces, customized built-ins allow engineers to inherit the accessibility and other user experience characteristics from HTML elements. Through the use of `ElementInternals`, form-associated custom elements enable engineers to develop feature-rich custom implementations of elements that can participate in HTML forms. In the final chapter, you used Declarative Shadow DOM to server-side render custom elements. Declarative Shadow DOM allows Shadow DOM to be declared entirely in a HTML template, opening up the possibility for custom elements to be statically rendered.

Specification

One of the potential hazards developing against browser specifications like custom elements v1 is that modern browsers implement specifications at different times, or potentially not at all. Such is the case with the customized built-in elements specification which the WebKit team has vehemently rejected[86]. While customized built-ins may never be included in Apple Safari, a polyfill exists[87] that brings the specification to the Safari browser. Polyfills patch functionality that doesn't exist in a web browser yet, while ponyfills provides that functionality in a module that usually executes after DOM has loaded. A ponyfill needs to statically analyze DOM to function, while a polyfill grants the functionality to the browser before DOM is parsed. Both ponyfills and polyfills are useful for custom element development as different specifications are introduced in web browsers over time.

Specifications can come and go. As browser vendors and members of the community propose new specifications, a browser may introduce a specification while others linger. One of the critiques of Web Components has been that specifications are unstable. While that may have been true during custom elements v0, a legacy specification that is not covered in this book, custom elements v1 is here to stay. The specifications engineers should be concerned with moving forward are those that augment the experience of developing custom elements. To illustrate how to navigate this terrain of web specifications, let's look at how CSS specifications have either been successful or failed to catch on.

As we observed in the exercises of this book, CSS variables can pierce the shadow boundary and help style custom elements. It would be rather limiting if CSS variables were the only tool for styling custom elements. Early in the development of the custom elements specifications, other CSS selectors were proposed that

[86] https://github.com/WICG/webcomponents/issues/509#issuecomment-224212371
[87] https://github.com/ungap/custom-elements#readme

pierced the shadow boundary (>>>, /deep/, and ::shadow), but they failed to gain traction as a standard. These selectors would have enabled a custom element to style slotted descendants through a special CSS selector. CSS Shadow Parts[88] were designed, in part, to address the concerns over implementing these aforementioned selectors in various browsers. CSS Shadow Parts allow engineers to declare parts of a HTML template that can be styled outside of the custom element's shadow boundary. While CSS Shadow Parts help CSS authors style a component from the outside, which could be useful for theming or styling a child element from a parent, this specification still doesn't address a problem with styling custom elements: the duplication of CSS styles throughout the codebase. Another specification named Constructable Stylesheets[89] aims to solve this problem by giving engineers a way to compose CSS for a custom element without the need to create a CSS stylesheet. At the time of writing this, CSS variables and CSS Shadow Parts are available in every mainstream browser, while Constructable Stylesheets are only found in Google Chrome, Microsoft Edge and behind a flag in Mozilla Firefox. This bodes well for the specification. Most browsers have signaled Constructable Stylesheets will be implemented and for those browsers which lag, a polyfill exists[90] that brings the functionality to that browser. Over time, the specifications that support styling custom elements have evolved, giving engineers greater flexibility when authoring elements that employ Shadow DOM.

It can often be confusing determining if a particular specification can be used in a specific browser. The Web Platform Incubator Community Group provides a spreadsheet[91] with the status of each specification. Chrome Platform Status[92] is also a useful tool for determining the viability of a particular specification. One source of

[88] https://www.w3.org/TR/css-shadow-parts-1/

[89] https://github.com/WICG/construct-stylesheets/blob/main/explainer.md

[90] https://github.com/calebdwilliams/construct-style-sheets

[91] https://docs.google.com/spreadsheets/d/1K2EtkvKXMDk_h2goR34uMIWmw1LfPhIi-QAJHcQBP_4/edit#gid=0

[92] https://chromestatus.com/features

confusion happens when a browser fails to signal if they intend to implement a specification, as is often the case with Apple Safari. Searching for a particular specification's polyfill may also yield information as the maintainers of the polyfill or ponyfill are keenly aware of browser support.

Another promising specification includes the HTML Template Instantiation Proposal[93] which brings data binding to HTML templates. A polyfill[94] exists to support these "Template Parts". The Context API Proposal[95] serves to provide a means to pass data between components. If adopted by mainstream browsers, this specification would replace the need for `BroadcastChannel` which we used in this book for the same purpose. The Pending Task Protocol Proposal[96] would alleviate issues with custom elements that have asynchronous dependencies. While a pattern for hydration exists as part of the implementation of Declarative Shadow DOM, Partial Hydration[97] would provide finer grained control over what should trigger hydration of a custom element. All of these specifications are indeed very promising. While some enterprising engineers may develop a polyfill when it doesn't exist yet, the rest of us should check the status of a specification, if a polyfill or ponyfill exists before developing against the specification.

Benefits

While navigating the browser landscape can be daunting, it shouldn't stop anyone from coding with custom elements today. The benefits of coding with browser specifications far outweigh

[93] https://github.com/WICG/webcomponents/blob/159b1600bab02fe9cd794825440a98537d53b389/proposals/Template-Instantiation.md

[94] https://github.com/github/template-parts

[95] https://github.com/webcomponents-cg/community-protocols/blob/main/proposals/context.md

[96] https://github.com/webcomponents-cg/community-protocols/blob/main/proposals/pending-task.md

[97] https://github.com/webcomponents-cg/community-protocols/issues/30

the potential negatives. By developing a UI library with custom elements, you can ensure the components have a long lifespan. Custom elements v1 will remain in the browser without breaking changes for the foreseeable future. Unlike framework-based UI components, custom elements don't incur the cost of dependency management. This can be a boon for large enterprise organizations that can struggle to maintain dependencies across disparate teams. When developing a UI library with React, for instance, teams must maintain the version of `react` and `react-dom` packages, along with any CSS-in-JS library. This can be disruptive at large corporations. When a corporate UI library upgrades various dependencies, teams using that library for use in their respective applications must upgrade as well. When Web Components are distributed in a fashion that requires zero dependencies, it can reduce the churn around maintaining dependencies at large enterprise organizations.

When UI libraries are distributed with Web Components, it allows a large corporation to scale. Web Components are interoperable with any JavaScript framework. Components that are interoperable can have a farther reach than components built with a JavaScript framework. Teams in large enterprise organizations adopt different JavaScript frameworks, having competing priorities which makes front end architectures difficult to maintain. If a UI library were distributed with framework-based UI components, it could be a blocker for teams. Angular components aren't compatible with React, for instance. If a UI library is distributed with Web Components, it removes barriers for adoption.

One benefit of Web Components we didn't explore in this book is how custom elements are composable. Custom elements are ES2015 classes that extend either from `HTMLElement`, or in the case of customized built-ins, the element that you wish to replicate some behavior. Once you've declared functionality in a component `class`, you could further extend that class and compose further functionality. With the micro-library you developed in this book,

you could even override the template or styling.

If the long lifespan, reusability, interoperability, and composability of custom elements weren't enough of a benefit, consider the performance implications. Custom elements have zero dependency, meaning no JavaScript is blocking the page from loading. Unlike most JavaScript frameworks which have to parse the DOM tree, custom elements are parsed directly by the browser. Declarative Shadow DOM allows custom elements to be statically rendered server-side, allowing blazingly fast page loads.

The future is bright

I wrote this book to help anyone who wants to learn the fundamentals of web development. At the component level that fundamental knowledge resides in the set of specifications known as Web Components. By engaging with the exercises in this book, I hope you've learned how to develop with custom elements in a meaningful way. Take what you've learned and apply it to developing a UI library where you work or if you want, follow the best practices you've learned for personal use.

Over the time I've been a software engineer, I've watched large enterprises both benefit and struggle with the adoption of JavaScript frameworks. The benefit of JavaScript frameworks is undeniable when developing an application, however; frameworks can fall short in certain ways when corporations need to scale their product. UI libraries help organizations scale since engineers across the company can reuse the UI components. JavaScript frameworks can't guarantee components built with the framework are interoperable with other JavaScript frameworks. Custom elements are interoperable, giving them a clear advantage over framework-based UI components. With additional specifications becoming available for custom element development sometime in the future, software

engineers will gain browser specifications that provide features once only available in JavaScript frameworks.

Custom elements don't just help large corporations scale, but they also help solve another problem in the web development community. Over the past few years, web development education has often focused on teaching software engineers the most popular JavaScript library: jQuery and then React. If higher education bypasses the fundamentals of web development: HTML, CSS, and JavaScript and teaches about frameworks instead, software engineers often don't fully understand how the web works "under the hood". When custom elements are developed using browser specifications, software engineers are immersed in HTML, CSS and JavaScript. By learning these fundamental technologies before a framework, it enables software engineers to diagnose problems and innovate new tools for web development. Web Components have emerged as a fundamental part of the web developer's toolkit and should be included in any web development curricula.

Web Components have been growing in popularity among web developers ever since Custom Elements v1 became available in every evergreen browser: Chrome, Safari, Firefox, and Edge. According to Google, in 2019 "> 5% of page loads use a Web Component". Chrome Platform Status[98] in 2022 revealed ~18% of page loads tracked by Google contain a Web Component. There is no better time than now to implement custom elements. I can't wait to see what you make with Web Components. Mention me in a tweet[99] or post in the Web Components Community on Twitter[100] and I'll take a look.

Special thanks to Westbrook Johnson[101] for providing insightful feedback during early stages of the writing process and my wife Susanne whose support made this book possible.

[98] https://chromestatus.com/metrics/feature/timeline/popularity/1689
[99] https://twitter.com/iplayitofflegit
[100] https://twitter.com/i/communities/1509750356040994816
[101] https://westbrookjohnson.com

Changelog

Revision 1 (04-04-2022)

Initial pre-release of the book

Revision 2 (12-06-2022)

Change book cover