



Foundation Dynamic Web Pages with Python

Create Dynamic Web Pages with
Django and Flask

David Ashley

Apress®

Foundation Dynamic Web Pages with Python

**Create Dynamic Web Pages
with Django and Flask**

David Ashley

Apress®

Foundation Dynamic Web Pages with Python

David Ashley
Austin, TX, USA

ISBN-13 (pbk): 978-1-4842-6338-9
<https://doi.org/10.1007/978-1-4842-6339-6>

ISBN-13 (electronic): 978-1-4842-6339-6

Copyright © 2020 by David Ashley

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6338-9. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

*This book is dedicated to Debbie and Jim. A brother never
had a better sister and brother-in-law.*

Table of Contents

About the Author	ix
About the Technical Reviewer	xi
Acknowledgments	xiii
Introduction	xv
Preface: Document Conventions.....	xvii
 Chapter 1: Introduction to Web Servers	 1
Glossary of Terms.....	1
The Apache Web Server	2
Nginx Web Server.....	5
Apache Tomcat Server	6
Configuring the Apache Web Server	6
Organizing Your Web Server.....	26
Operating System Links.....	26
Apache Directives.....	27
Summary.....	27
 Chapter 2: HTML Pages and CSS.....	 29
HTML Tags.....	29
Document and Metadata Tags	31
The Text Tags	32
Grouping Content Tags.....	33
Sectioning Tags	34

TABLE OF CONTENTS

Creating Tables	35
Creating Forms	36
Embedding Content Tags	37
CSS Elements.....	38
CSS Selectors	40
Border and Background Properties	43
Box Model Properties	46
Text Properties.....	47
Transition, Animation, and Transform Properties	48
Other Properties	50
Organizing HTML and CSS Documents	50
Simple HTML and CSS Pages	52
A Complete HTML and CSS Page	57
Creating a Library of HTML Page Segments	65
Summary.....	69
Chapter 3: Using CGI and Python	71
Your First cgi-bin Program	71
CGI Program Strategy.....	75
Setting Up the html_lib Parts	76
A Portable and Maintainable CGI Program	78
More Partial HTML Skeletons.....	84
Hyperlink HTML Skeletons.....	84
A More Complicated Hyperlink Example	94
Calling and Passing Data to a CGI Program	100
The GET Method	101
The POST Method	107

Another POST Method with HTML Text Data.....	112
Using POST with a dropdown Box	115
Cookies in CGI	117
Sending Cookies from the Server.....	117
Retrieving Cookies.....	119
Summary.....	120
Chapter 4: Using SSI and Python	123
Getting Started.....	123
The config SSI Directive.....	125
The echo SSI Directive	129
The exec SSI Directive	130
The fsize SSI Directive	135
The flastmod SSI Directive.....	138
The include SSI Directive	140
Additional SSI Directives.....	142
The SSI set Directive	142
The SSI Conditional Directives.....	147
Summary.....	158
Chapter 5: Using Flask and Jinja	159
WSGI and Flask.....	160
Installing and Testing WSGI	160
Installing and Testing Flask	166
Jinja2	171
Summary.....	181

TABLE OF CONTENTS

Chapter 6: Django	183
Django and WSGI.....	183
Configuring and Testing Django	185
Django and Templates.....	191
Using a Database with Django	198
Summary.....	199
Chapter 7: Comparing CGI, SSI, Flask, and Django	201
Installation and Configuration	201
CGI Installation and Configuration	202
SSI Installation and Configuration	202
Flask Installation and Configuration	203
Django Installation and Configuration	203
Python Usage	204
CGI Python Usage	204
SSI Python Usage	205
Flask Python Usage	205
Django Python Usage.....	206
Template Processing.....	206
CGI Template Processing	206
SSI Template Processing	207
Flask Template Processing	207
Django Template Processing	207
Database Access.....	207
Rest APIs	208
Summary.....	208
Index.....	209

About the Author



David Ashley is a technical writer for SkillSoft where he specializes in open source, particularly Linux. As a member of the Linux Fedora documentation team he recently led the Libvirt project documentation and wrote the Python programs included with it. He has developed in 20 different programming languages during his 30 years as a software developer and IT consultant, including more than 18 years at IBM and 12 years with American Airlines.

About the Technical Reviewer



Akshay Saini is Mumbai-based tech book reviewer with several years' experience in IT as a software developer specializing in Python.

Recently he has developed new product solutions for the media and entertainment industry for clients such as Olympic Broadcasting Services (OBS), ViacomCBS, and Sportcast.

Reviewing and writing a tech book has always been on his bucket list, and with this book it became a reality.

He is a tech lover and spends much time developing software to build a better future for society.

Acknowledgments

I would like to acknowledge the people who contributed their time and efforts to this book. There are just too many to list here, but I would especially like to acknowledge all the people at Apress who helped put this book together.

Introduction

This book compares some of the best-known dynamic HTML page creation systems. It includes some older systems such as CGI and SSH as well as newer systems such as Flask and Django. Each system is examined and compared with the other systems to discover each of their strengths and weaknesses.

This should give you a basis for choosing the correct system for your dynamic HTML page needs. For each system, I will provide example programs so that if you are not experienced with a system, you can get a taste of what building an application with it is like.

Preface: Document Conventions

This book uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information. The convention used depends on the type of information displayed.

Computer Commands

Computer commands are usually presented in a bold font such as in the following example:

The Unix command **ls** run from the command shell is used to present a list of files and directories.

Filenames

Filenames are usually presented in monospaced text such as in the following example:

To see the contents of the file `report.txt` use the command `cat report.txt`.

Programming Language Elements and Literals

Programming language elements include such things variable names, literals, constants, symbols, tokens, functions, class names, and other objects.

Literal data is taken directly from a computer screen or a computer language literal value and is usually presented in monospaced text such as within the following example:

The following line is the output from running **ls**:

```
en-US Makefile publican.cfg
```

Computer Output and Source Code

Computer output data is taken directly from a computer screen and usually presented in monospaced text such as in the following example. This information is usually set off from the rest of the text.

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set off from the rest of the text, as shown in Listing [P-1](#).

Listing P-1. This Is a Source Code Listing

```
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

 **Note** Notes are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

 **Important** Important boxes detail things that are easily missed: configuration changes that apply only to the current session, or services that need restarting before an update will apply. Ignoring a box labeled “Important” will not cause data loss but may cause irritation and frustration.

 **Warning** Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

CHAPTER 1

Introduction to Web Servers

This chapter introduces web servers, the services they provide, and how they work. This information is essential to web developers so they can make proper use of their web server and provide the best web experience to their users.

All web servers use the same building blocks to serve up web pages to the user. While we could look at all the available web servers, it really is not necessary since they all are designed around the same building blocks. Instead, we will concentrate on the Apache web server since it is the most popular. All the other web servers use the same building blocks and design principles as the Apache server.

Glossary of Terms

When delving into technical information, it is important that you understand the terminology used. For that reason, please review the following web server terms:

- *Common Gateway Interface (CGI)*: This describes a process that serves up a dynamic web page. The web page is built by a program provided by the web server

administrator. A common set of information is available to the program via the program's environment.

- *Hypertext Transport Protocol (HTTP)*: This is a set of rules used to describe a request by the user to the web server and the returned information. The request and the reply must follow strict rules for the request to be understood by the server and for the reply to be understood by the user's browser.
- *Hypertext Markup Language (HTML)*: This code is used to build a web page that is displayed by a browser, and the Apache web server is used to serve the web page to users (clients). There are several versions of this code, but we will be using the latest version (5.0) in this book.
- *Cascading Style Sheets (CSS)*: These sheets define the styles to be used by one or more web pages. These styles are used to define fonts, colors, and the size of a section of text within a defined area of the HTML page.

These terms should give you a good starting point for discussing how a web server works. All of these terms will receive wider attention and definition throughout this book.

The Apache Web Server

The Apache web server (today this is known formally as the HTTP Server) dates back to the mid-1990s when it started gaining widespread use. The web server is a project of the Apache Software Foundation, which manages several projects. There are currently more than 200 million lines of code managed by the foundation for the Apache web server. The current release as of this writing is 2.4.41.

Starting with Apache version 2.0, Apache uses a hook architecture to define new functionality via modules. We will study this in a subsequent chapter.

When you first look at hooks, they will seem a little complicated, but in reality, they are not since most of the time you are only modifying Apache a little. This will reduce the code you need to write to implement a hook to a minimum.

The Apache web server uses a config file to define everything the server needs to know about all the hooks you want to include in Apache. It also defines the main server and any virtual servers you want to include. In addition, it defines the name of the server, the home directory for the server, the CGI directory to be used, any aliases needed by the server, the server name, any specific handlers used by that server, the port to be used by the server, the error log to be used, and several other factors.

Once configuration is complete, the Apache server is now ready to supply files to a client browser. This is called the *request-response cycle*. For each request sent by the browser to the server, the request must travel through the request-response cycle to produce a response that is sent back to the browser. While this looks simple on the surface, the request-response cycle is both powerful and flexible. It can allow programs you create, called *modules*, to modify both the request and the response in many flexible ways. A module can also create the response from scratch and can include inputs from resources outside of Apache, such as a database or other external data repository.

Figure 1-1 shows the request-response loop of Apache plus the startup and shutdown phases of the server.

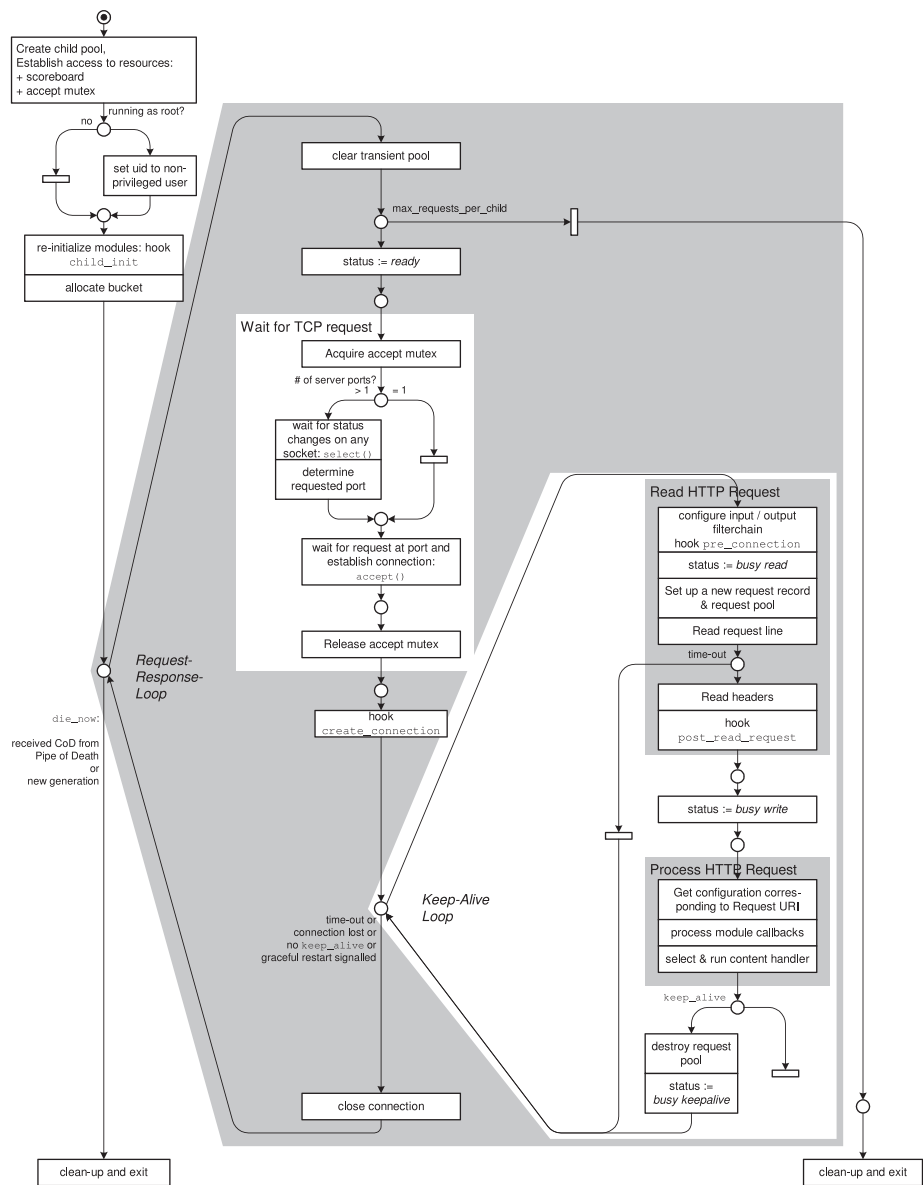


Figure 1-1. The Apache request-response loop

Modules not only can be used in the request-response cycle but in other portions of Apache such as during configuration, shutdown/cleanup, processing security requests, and other valuable functions. As you can see, modules allow flexible and powerful methods to be created by the server administrator to help with providing a great experience for their users.

Modules are not the only way to create dynamic web pages. Another way is by invoking available Apache services that can call an external program to create the page. The CGI process is usually invoked to supply this service, but there are other ways as well. Each of these ways will be examined in this book. It will be up to you to decide the best methodology for use in your environment.

The shaded request/response loop can have several forms. One such form is as a loop inside one of several processes under Apache. Another is running the loop as a thread inside a single process under Apache. All of these forms are designed to make the most efficient process of responding to a request that an operating system may provide.

The Keep-Alive loop is for HTTP 2.0 requests if supported by the web server. It allows the connection to stay open to the client until all requests have been processed. The loop here describes how a single request is processed by Apache. If the web server is not running HTTP 2.0 requests, then each request/response will close the connection once the response has been sent.

Nginx Web Server

The Nginx server was designed as a low-cost (in terms of system requirements) alternative to the Apache server. Probably the biggest difference between Nginx and Apache is that Nginx has an asynchronous event-driven architecture rather than using multiple threads to process each request. While this can provide predictable performance under high loads, it does come with some downsides. For instance, a request can end

up waiting in the queue longer than the request attempt will survive on the network; i.e., the requester can give up before the request is ever processed if there are too few processing routines. While this problem is not exclusive to this server, it does still exist.

Recently the Nginx server has become popular within the community because of its smaller footprint and flexible design. However, since many of the principles that we will use to describe the Apache server also apply to the Nginx server, I will not delve deeply into Nginx and will discuss it only when differences between the two servers are important, especially in regard to dynamic web page design.

Apache Tomcat Server

The Apache Tomcat server is written in Java, which makes it difficult to compare to the more standard web servers. While some principles of dynamic web page design are similar, there are many differences. Therefore, and because it's less commonly used than Apache and Nginx, I will not attempt to cover it in this book.

Configuring the Apache Web Server

The Apache web server has a single main configuration file and a number of optional configuration files. The main file is named `httpd.conf`, and it controls which optional files are loaded as well as the location where they can be found. It also specifies the global features used by the server.

Listing 1-1 shows an unedited version of the `httpd.conf` file. Following the listing, I will describe the sections that need to be modified to give you a usable configuration file.

Listing 1-1. The Unedited httpd.conf File

```
#
# This is the main Apache HTTP server configuration file.
#   It contains the
# configuration directives that give the server its
#   instructions.
# See <URL:http://httpd.apache.org/docs/2.4/> for detailed
#   information.
# In particular, see
# <URL:http://httpd.apache.org/docs/2.4/mod/directives.html>
# for a discussion of each configuration directive.
#
# See the httpd.conf(5) man page for more information on this
#   configuration,
# and httpd.service(8) on using and configuring the httpd service.
#
# Do NOT simply read the instructions in here without
#   understanding
# what they do.  They're here only as hints or reminders.
#   If you are unsure
# consult the online docs.  You have been warned.
#
# Configuration and logfile names: If the filenames you specify
#   for many
# of the server's control files begin with "/" (or "drive:/"
#   for Win32), the
# server will use that explicit path.  If the filenames do
#   *not* begin
# with "/", the value of ServerRoot is prepended -- so
#   'log/access_log'
```

CHAPTER 1 INTRODUCTION TO WEB SERVERS

```
# with ServerRoot set to '/www' will be interpreted by the
# server as '/www/log/access_log', where as '/log/access_log'
# will be
# interpreted as '/log/access_log'.

#
# ServerRoot: The top of the directory tree under which the
# server's
# configuration, error, and log files are kept.
#
# Do not add a slash at the end of the directory path. If you point
# ServerRoot at a non-local disk, be sure to specify a local
# disk on the
# Mutex directive, if file-based mutexes are used. If you wish
# to share the
# same ServerRoot for multiple httpd daemons, you will need to
# change at
# least PidFile.
#
ServerRoot "/etc/httpd"

#
# Listen: Allows you to bind Apache to specific IP addresses
# and/or
# ports, instead of the default. See also the <VirtualHost>
# directive.
#
# Change this to Listen on specific IP addresses as shown below to
# prevent Apache from glomming onto all bound IP addresses.
#
#Listen 12.34.56.78:80
Listen 80
```

```

#
# Dynamic Shared Object (DSO) Support
#
# To be able to use the functionality of a module which was
# built as a DSO you
# have to place corresponding 'LoadModule' lines at this
# location so the
# directives contained in it are actually available _before_
# they are used.
# Statically compiled modules (those listed by 'httpd -l') do
# not need
# to be loaded here.
#
# Example:
# LoadModule foo_module modules/mod_foo.so
#
Include conf.modules.d/*.conf

#
# If you wish httpd to run as a different user or group, you
# must run
# httpd as root initially and it will switch.
#
# User/Group: The name (or #number) of the user/group to run
# httpd as.
# It is usually good practice to create a dedicated user and
# group for
# running httpd, as with most system services.
#
User apache
Group apache

```

CHAPTER 1 INTRODUCTION TO WEB SERVERS

```
# 'Main' server configuration
#
# The directives in this section set up the values used by the 'main'
# server, which responds to any requests that aren't handled by a
# <VirtualHost> definition. These values also provide defaults for
# any <VirtualHost> containers you may define later in the file.
#
# All of these directives may appear inside <VirtualHost>
# containers,
# in which case these default settings will be overridden for the
# virtual host being defined.
#
#
# ServerAdmin: Your address, where problems with the server
# should be
# e-mailed. This address appears on some server-generated
# pages, such
# as error documents. e.g. admin@your-domain.com
#
ServerAdmin root@localhost
#
# ServerName gives the name and port that the server uses to
# identify itself.
# This can often be determined automatically, but we recommend
# you specify
# it explicitly to prevent problems during startup.
#
# If your host doesn't have a registered DNS name, enter its IP
# address here.
```

```
#
#ServerName www.example.com:80

#
# Deny access to the entirety of your server's filesystem. You must
# explicitly permit access to web content directories in other
# <Directory> blocks below.
#
<Directory />
    AllowOverride none
    Require all denied
</Directory>

#
# Note that from this point forward you must specifically allow
# particular features to be enabled - so if something's not
# working as
# you might expect, make sure that you have specifically
# enabled it
# below.
#
#
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this
# directory, but
# symbolic links and aliases may be used to point to other
# locations.
#
DocumentRoot "/var/www/html"

#
# Relax access to content within /var/www.
```

```
#
<Directory "/var/www">
    AllowOverride None
    # Allow open access:
    Require all granted
</Directory>

# Further relax access to the default document root:
<Directory "/var/www/html">
    #
    # Possible values for the Options directive are "None", "All",
    # or any combination of:
    #   Indexes Includes FollowSymLinks SymLinksifOwnerMatch
    #   ExecCGI MultiViews
    #
    # Note that "MultiViews" must be named *explicitly* ---
    # "Options All"
    # doesn't give it to you.
    #
    # The Options directive is both complicated and
    # important. Please see
    # http://httpd.apache.org/docs/2.4/mod/core.html#options
    # for more information.
    #
    Options Indexes FollowSymLinks

    #
    # AllowOverride controls what directives may be placed in
    # .htaccess files.
    # It can be "All", "None", or any combination of the
    # keywords:
    #   Options FileInfo AuthConfig Limit
    #
```

```

    AllowOverride None

    #
    # Controls who can get stuff from this server.
    #
    Require all granted
</Directory>

#
# DirectoryIndex: sets the file that Apache will serve if a
#   directory
#   is requested.
#
<IfModule dir_module>
    DirectoryIndex index.html
</IfModule>

#
# The following lines prevent .htaccess and .htpasswd files
#   from being
#   viewed by Web clients.
#
<Files ".ht*">
    Require all denied
</Files>

#
# ErrorLog: The location of the error log file.
# If you do not specify an ErrorLog directive within a
#   <VirtualHost>
#   container, error messages relating to that virtual host will be
#   logged here.  If you *do* define an error logfile for a
#   <VirtualHost>

```

CHAPTER 1 INTRODUCTION TO WEB SERVERS

```
# container, that host's errors will be logged there and not here.
#
ErrorLog "logs/error_log"

#
# LogLevel: Control the number of messages logged to the error_log.
# Possible values include: debug, info, notice, warn, error, crit,
# alert, emerg.
#
LogLevel warn

<IfModule log_config_module>
    #
    # The following directives define some format nicknames for use with
    # a CustomLog directive (see below).
    #
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
    \"%{User-Agent}i\"" combined
    LogFormat "%h %l %u %t \"%r\" %>s %b" common

    <IfModule logio_module>
        # You need to enable mod_logio.c to use %I and %O
        LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
        \"%{User-Agent}i\" %I %O" combinedio
    </IfModule>

    #
    # The location and format of the access logfile (Common
    # Logfile Format).
    # If you do not define any access logfiles within a
    # <VirtualHost>
    # container, they will be logged here. Contrariwise, if
    # you *do*
```

```

# define per-<VirtualHost> access logfiles, transactions will be
# logged therein and *not* in this file.
#
#CustomLog "logs/access_log" common

#
# If you prefer a logfile with access, agent, and referer
# information
# (Combined Logfile Format) you can use the following
# directive.
#
CustomLog "logs/access_log" combined
</IfModule>

<IfModule alias_module>
#
# Redirect: Allows you to tell clients about documents that
# used to
# exist in your server's namespace, but do not anymore. The client
# will make a new request for the document at its new location.
# Example:
# Redirect permanent /foo http://www.example.com/bar

#
# Alias: Maps web paths into filesystem paths and is used to
# access content that does not live under the DocumentRoot.
# Example:
# Alias /webpath /full/filesystem/path
#
# If you include a trailing / on /webpath then the server will
# require it to be present in the URL. You will also likely
# need to provide a <Directory> section to allow access to
# the filesystem path.

```

```
#
# ScriptAlias: This controls which directories contain
# server scripts.
# ScriptAliases are essentially the same as Aliases, except that
# documents in the target directory are treated as
# applications and
# run by the server when requested rather than as documents
# sent to the
# client. The same rules about trailing "/" apply to
# ScriptAlias
# directives as to Alias.
#
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"

</IfModule>

#
# "/var/www/cgi-bin" should be changed to whatever your
# ScriptAliased
# CGI directory exists, if you have that configured.
#
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options None
    Require all granted
</Directory>

<IfModule mime_module>
#
# TypesConfig points to the file containing the list of
# mappings from
# filename extension to MIME-type.
#
```

```

TypesConfig /etc/mime.types

#
# AddType allows you to add to or override the MIME
# configuration
# file specified in TypesConfig for specific file types.
#
#AddType application/x-gzip .tgz
#
# AddEncoding allows you to have certain browsers
# uncompress
# information on the fly. Note: Not all browsers support this.
#
#AddEncoding x-compress .Z
#AddEncoding x-gzip .gz .tgz
#
# If the AddEncoding directives above are commented-out,
# then you
# probably should define those extensions to indicate media types:
#
AddType application/x-compress .Z
AddType application/x-gzip .gz .tgz

#
# AddHandler allows you to map certain file extensions to
# "handlers":
# actions unrelated to filetype. These can be either built
# into the server
# or added with the Action directive (see below)
#
# To use CGI scripts outside of ScriptAliased directories:
# (You will also need to add "ExecCGI" to the "Options"
# directive.)
#

```

```

#AddHandler cgi-script .cgi

# For type maps (negotiated resources):
#AddHandler type-map var

#
# Filters allow you to process content before it is sent to
  the client.
#
# To parse .shtml files for server-side includes (SSI):
# (You will also need to add "Includes" to the "Options"
  directive.)
#
AddType text/html .shtml
AddOutputFilter INCLUDES .shtml
</IfModule>

#
# Specify a default charset for all content served; this enables
# interpretation of all content as UTF-8 by default. To use the
# default browser choice (ISO-8859-1), or to allow the META tags
# in HTML content to override this choice, comment out this
# directive:
#
AddDefaultCharset UTF-8

<IfModule mime_magic_module>
#
# The mod_mime_magic module allows the server to use
  various hints from the
# contents of the file itself to determine its type. The
  MIMEMagicFile

```

```

# directive tells the module where the hint definitions are
  located.
#
  MIMEMagicFile conf/magic
</IfModule>

#
# Customizable error responses come in three flavors:
# 1) plain text 2) local redirects 3) external redirects
#
# Some examples:
#ErrorDocument 500 "The server made a boo boo."
#ErrorDocument 404 /missing.html
#ErrorDocument 404 "/cgi-bin/missing_handler.pl"
#ErrorDocument 402 http://www.example.com/subscription_info.html
#
#
# EnableMMAP and EnableSendfile: On systems that support it,
# memory-mapping or the sendfile syscall may be used to deliver
# files. This usually improves server performance, but must
# be turned off when serving from networked-mounted
# filesystems or if support for these functions is otherwise
# broken on your system.
# Defaults if commented: EnableMMAP On, EnableSendfile Off
#
#EnableMMAP off
EnableSendfile on

# Supplemental configuration
#
# Load config files in the "/etc/httpd/conf.d" directory, if any.
IncludeOptional conf.d/*.conf

```

We will look at each directive used in this file individually starting with the `ServerRoot` directive.

```
ServerRoot "/etc/httpd"
```

This directive indicates the root where all the server's global files are located. This includes all the configuration files, the log files, and the error files.

The next directive is the `Listen` directive, and it specifies the TCP/IP port that the server will listen on.

```
#Listen 12.34.56.78:80
Listen 80
```

The `Listen` directive shown earlier specifies that port 80 should be used as the listening port on the default IP address.

The next directive specifies where the necessary global modules should be loaded from.

```
Include conf.modules.d/*.conf
```

This `Include` directive specifies the location of the configuration files that will load the global modules. What is specified here is a subdirectory off the `ServerRoot` directory. Only files in this subdirectory with an extension of `.conf` will be loaded and thus also the modules actually loaded from them. There are multiple configuration files because the modules are divided into different categories and can thus be included or excluded easily.

The next directives are the `User` and `Group` directives.

```
User apache
Group apache
```

The `User` directive specifies what `userID` Apache should run under, and the `Group` directive specifies what `groupID` it should run under. While this may not seem important, it actually is, especially when running more

than one server. Each server may need to be running under a different userID and groupID to work properly. If you are running multiple servers, then consider moving the userID and groupID to the individual server configuration files.

All the following directives set up the values used by the “main” server, which responds to any requests that aren’t handled by a <VirtualHost> definition. These values also provide defaults for any <VirtualHost> containers you may define later in the file.

All of these directives may appear inside <VirtualHost> containers, in which case these default settings will be overridden for the virtual host being defined.

The next directive specifies where you want problems with your server to be emailed.

```
ServerAdmin root@localhost
```

The ServerAdmin directive should be changed to an email address under the server administrator that has the authority to handle problems with the Apache server.

The next directive specifies the name known to the DNS server used by the machine running the Apache server.

```
ServerName www.example.com:80
```

You should note that not only the name should be specified but also the listening port. The port does not have to be unique if you are running multiple servers on the same TCP/IP address and port. For more information concerning this kind of configuration, you should refer to the Apache documentation.

The next section is extremely important!

```
<Directory />
    AllowOverride none
    Require all denied
</Directory>
```

Do not modify this section unless you know exactly what you are doing. My suggestion is to *never* modify it. This protects your root file system from snoopers by removing read access from all Apache users (not system users) to the root directory. If you intend to modify this section, you should consult with your security team first.

The next directive, `DocumentRoot`, specifies the root of all the files on your server, or at least most of them. There are some exceptions that we will run into later.

```
DocumentRoot "/var/www/html"
```

You can point the `DocumentRoot` directive anywhere you want on your file system, but it is suggested that it should always be local within the root file system so that during startup of the server this location will always be visible.

The next section relaxes access to the content of the `DocumentRoot`:

```
<Directory "/var/www">
    AllowOverride None
    # Allow open access:
    Require all granted
</Directory>
```

The previous `DocumentRoot` directive pointed our server to the root of our document file system. Now we have to relax the access to that point so the server can have access to the files. This section of code gives all access rights to the server.

We actually need additional access rights, so the following gives us those rights:

```
Options Indexes FollowSymLinks
AllowOverride None
```

The `Options` directive in this case allows a listing of all the files to be sent to the user just in case an `index.html` file does not exist. It also

forces the server to follow symbolic links even if they point outside the DocumentRoot tree.

The next directive is used to specify the name of the file that will be fetched should one not be specified by the user.

```
DirectoryIndex index.html
```

In this case, the name of the file is index.html.

The following lines prevent the .htaccess and .htpasswd files from being viewed by clients:

```
<Files ".ht*">
    Require all denied
</Files>
```

The following directive specifies the name and location of the error log:

```
ErrorLog "logs/error_log"
```

When the server needs to log an error, this is where it will be placed.

Next, we want to at least control what errors get logged.

```
# Possible values include: debug, info, notice, warn, error, crit,
# alert, emerg.
#
LogLevel warn
```

The possible values here are debug, info, notice, warn, error, crit, alert, and emerg in order from lowest to highest. Only the level you specify plus higher levels will be logged; all lower levels will be ignored.

The next section contains several directives specifying the format that log messages will appear in:

```
<IfModule log_config_module>
```

The formats are specified in C's `sprintf` format. These formats are relatively straightforward and easily understood. Also specified in this section are the name and location of the access log. The directives in this section are executed only if the corresponding module is loaded previously.

The next section uses the Alias module to further alias other subdirectories.

```
<IfModule alias_module>
```

This module created fake entries in the Apache document directory tree, which can be referred to by the user. An example of this is the `/cgi-bin` subdirectory, which can hold programs to be executed by the user's browser to produce a dynamic HTML page. The directives in this section are executed only if the corresponding module is loaded previously.

The following section sets the characteristics for the `/cgi-bin` subdirectory:

```
<Directory "/var/www/cgi-bin">
```

The directives in this section make whatever programs are within the `/cgi-bin` subdirectory executable and readable for the user browser.

The next section adds default actions for certain file types that can be executed by the browser if it is supported by the browser.

```
<IfModule mime_module>
```

There are a number of actions based on file type(s) specified here. The directives in this section are executed only if the corresponding module is loaded previously.

When the server provides output to the browser, we need a default character set to use in case the HTML page does not specify one.

```
AddDefaultCharset UTF-8
```

The previous directive sets the UTF-8 character set as the default character set to be used by all web pages when no character set is explicitly set within the web page.

The next section gives hints to the server about various file types:

```
<IfModule mime_magic_module>
```

The `mod_mime_magic` module allows the server to use various hints from the contents of the file itself to determine its type. The directives in this section are executed only if the corresponding module is loaded previously.

The next section enables the operating system to return a file to the client:

```
#EnableMMAP off
EnableSendfile on
```

On systems that support it, memory-mapping or the `sendfile` syscall may be used to return files to the browser. This usually improves server performance but must be turned off when serving from a networked-mounted file system or if support for these functions is otherwise broken on your system.

Lastly, additional configuration files may be included.

```
IncludeOptional conf.d/*.conf
```

If the Apache server is supporting one or more virtual machines, then this location will usually contain a file for the configuration of each virtual machine. All the directives usually contained in one of these files have been shown previously, although you may find they contain additional directives to support functions needed by one or more virtual machines.

As you can see, Apache supports many directives to configure the server, many of which have not been discussed in this chapter. A really good Apache reference manual is required to make full use of the power of the server. Later in this book we look at some additional directives for use in our discussion of dynamic web pages.

Organizing Your Web Server

This topic, while at first glance appears to be simple, is actually fraught with potential problems of your own making. It is so easy to fall into traps when organizing a website that this topic deserves special treatment.

The problems that you can create for yourself all revolve around security. In the previous section, the default configuration points the DocumentRoot to a location that in the beginning looks pretty safe because it is mostly empty. The problem comes when you point the DocumentRoot to some other spot on your file system. Apache has no idea if you have just created a problem for yourself.

We need to examine the potential problems that you can make for yourself both now and in the future.

Operating System Links

Operating system links (created with the `ln` command) create a shortcut to another part of the file system. While this may seem like a nice way to build certain parts of your file system for use by Apache, you should be extremely careful when doing so. Not only can links bring in some features you would otherwise have to duplicate, but they can also bring in unintended files that either might be sensitive from a security point of view or not readable by the Apache user (that can be good or bad depending on the circumstances).

Remember also that when you bring other parts of the file system into Apache's root, you may or may not have administrative privileges on that part of the file system. This means others can make changes to the file system that might have an adverse impact on your expectations.

Ownership of files being brought into Apache's purview can also be a problem. Apache may not be able to read these files, so that needs to be evaluated for your circumstances. Also, ownership of these files can

change at any time, and that may have a negative impact on the usability of your site.

Lastly, new files may be added at any time, and older files may be deleted at any time. You may or may not get notifications about these additions/deletions, but they could have impacts on your website and need to be evaluated *before* they occur, not after.

Apache Directives

The Apache directives `Alias` and `ScriptAlias` are similar to the operating system `ln` command in that they bring in other parts of the local file system to the file system created by Apache. The only real difference is that the `Alias` and `ScriptAlias` directives are logical and not physical; i.e., only Apache can see them, and they do not really exist in the file system. But from the perspective of the web administrator, they have the same potential problems as real links. All the problems listed in the previous section also apply to these directives.

Summary

This chapter introduced you to web servers in general with specific focus on the Apache web server. I also introduced you to some web server concepts as well as some topics specific to Apache.

I introduced you to Apache's configuration file and described the required and optional features available to the administrator when configuring Apache.

Lastly, I introduced you to some potential problems you may run into when organizing your website's file system.

CHAPTER 2

HTML Pages and CSS

This chapter is all about Hypertext Markup Language (HTML) and the corresponding Cascading Style Sheets (CSS) language. This is just an introduction, and it is highly recommended that you obtain specialized references on both of these topics to gain a complete understanding of how these languages work together to present information to the user.

The HTML language has a long history going back to the 1970s. It is what is known as a *tag language*. It uses special tags within the text within your HTML page to identify different kinds of text. There are tags to identify paragraphs, numbered lists, definition lists, headings, tables, forms, images, and many other parts.

The CSS language actually comes in two forms. The first form of the language is attached to HTML tags as attributes. The second form of the language is as a stand-alone document that completely describes how the document is to be formatted. We will describe this in more detail later in this chapter.

HTML Tags

HTML tags have a long history going back to the Generalized Markup Language (GML) tag system invented by IBM. GML was the result of modernizing an earlier language that described both the formatting of a document and the style of the document. In the early days of printing hardware, there was not much differentiation between the printer and the

formatting of a document. But as the number and variations of printers increased, the need to separate the style of the document from identifying the elements of a document increased dramatically. By the beginning of the 1980s, there were hundreds of different kinds of printers including character-, line-, and laser-based. This increase really exploded during the 1980s.

This explosion of printing hardware required a rethinking of content versus style. In this way, a style can be tuned to an output device while the content remains unchanged. The IBM GML language evolved during the 1980s into the Standardized General Markup Language (SGML) tag language. This was a relatively short-lived standard as it was hard to learn, extensive, and practically unusable to write documents of any kind. But it did become the basis for two child languages: HTML and XML. These two languages used many of the concepts pioneered in SGML and left out anything not really needed for their purpose.

HTML has gone through several revisions since its first release in the early 1990s. Currently, we are at version 5.0 of HTML. This version has taken on a few characteristics of the XML language and added comprehensive use of CSS to form a language that is easy to use and code for even neophytes. The number of tags has been kept to a minimum, and they are easy to remember. There are any number of reference books to help you learn to code your documents with HTML and CSS, but they have a limitation in that they usually show you how to write relatively simple HTML pages. What they do not concentrate on is how to write dynamic web pages, i.e., pages that can change based on input from the user or the data itself.

Although CSS is undeniably integral to HTML, this book will only cover the basics of that integration. Any CSS reference will cover the same material in more detail, and I will leave that to you to learn.

Document and Metadata Tags

Table 2-1 summarizes the document and metadata tags. These tags are used to describe the superstructure of your document's HTML page. These tags provide information to the browser concerning the document, define the styles, and specify any scripts used by the document.

Many of the descriptions were taken from other sources on the Web and elsewhere. Any mistakes in the text are mine alone.

Table 2-1. *The Document/Metadata Tags*

Tag	Description	Type
<base>	Sets the base for all relative URLs.	Metadata
<body>	Denotes the contents of the document.	N/A
<DOCTYPE>	The first tag that should appear in your document. It denotes the document type to identify it to the browser. This not a real HTML tag, just an identifier.	N/A
<head>	Contains the document metadata.	N/A
<html>	Indicates the start of HTML in a document.	N/A
<link>	Defines a relationship with an external resource.	Metadata
<meta>	Provides information about the document.	Metadata
<noscript>	Contains content that will be displayed when scripting is disabled or unavailable in the browser.	Metadata/phrasing
<script>	Defines a script block, either inline or in an external file.	Metadata/phrasing
<style>	Defines a CSS style.	Metadata
<title>	Sets the title for the document.	Metadata

The Text Tags

The text tags are applied to content to give basic structure and meaning. Table 2-2 summarizes these tags.

Table 2-2. *The Text Tags*

Tag	Description	Type
<a>	Creates a hyperlink.	Phrasing/flow
<abbr>	Denotes an abbreviation.	Phrasing
	Offsets a span of text without additional emphasis or importance.	Phrasing
 	Denotes a line break.	Phrasing
<cite>	Denotes the title of another work.	Phrasing
<code>	Denotes a fragment of computer code.	Phrasing
<del>	Denotes text that has been removed from the document.	Phrasing/flow
<dfn>	Denotes the definition of a term.	Phrasing
	Denotes a span of text with emphasis.	Phrasing
<style>	Defines a CSS style.	Metadata
<i>	Denotes a span of text that is of a different nature than the surrounding content, such as a word from another language.	Phrasing
<ins>	Denotes text that has been inserted into the document.	Phrasing/flow
<kbd>	Denotes user input.	Phrasing
<mark>	Denotes content that is highlighted because of its relevance in another context.	Phrasing

(continued)

Table 2-2. *(continued)*

Tag	Description	Type
<q>	Denotes content quoted from another source.	Phrasing
<rp>	Denotes parameters for use with the <ruby> tag.	Phrasing
<rt>	Denotes a notation for use with the <ruby> tag.	Phrasing
<ruby>	Denotes notation to be placed above or to the right of characters in a logographic language.	Phrasing
<s>	Denotes text that is no longer accurate.	Phrasing
<samp>	Denotes output from a computer program.	Phrasing
<small>	Denotes fine print.	Phrasing
	A generic tag that does not have any semantic meaning of its own. Use this tag to apply global attributes without imparting additional semantic significance.	Phrasing
	Denotes text that is important.	Phrasing
<sub>	Denotes subscript text.	Phrasing
<sup>	Denotes superscript text.	Phrasing
<time>	Denotes a time and/or date.	Phrasing
<u>	Offsets a span of text without additional emphasis or importance.	Phrasing
<var>	Denotes a variable from a program or computer system.	Phrasing
<wbr>	Denotes a place where a line break can be safely placed.	Phrasing

Grouping Content Tags

The tags in Table 2-3 are used to associate related content into groups.

Table 2-3. *The Grouping Tags*

Tag	Description	Type
<blockquote>	Denotes a block of content quoted from another source.	Flow
<dd>	Denotes a definition with a <dl> tag.	N/A
<div>	A generic element that does not have any predefined semantic significance. This is the flow equivalent of the tag.	Flow
<dl>	Denotes a description list that contains a series of terms and definitions.	Flow
<dt>	Denotes a term within the <dl> tag.	N/A
<figcaption>	Denotes a caption for a <figure> tag.	N/A
<figure>	Denotes a figure.	Flow
<hr>	Denotes a paragraph-level thematic break.	Flow
	Denotes an item in a , , or <menu> tag.	N/A
	Denotes an ordered list of items.	Flow
<p>	Denotes a paragraph.	Flow
<pre>	Denotes content whose formatting should be preserved.	Flow
	Denotes an unordered list of items.	Flow

Sectioning Tags

The tags in Table 2-4 are used to break down the content so that each concept, idea, or topic is isolated.

Table 2-4. *The Section Tags*

Tag	Description	Type
<address>	Denotes contact information for a document or article	Flow
<article>	Denotes an independent block of content	Flow
<aside>	Denotes content that is tangentially related to the surrounding content	Flow
<details>	Creates a section the user can expand to get additional details	Flow
<footer>	Denotes a footer region	Flow
<h1-h6>	Denotes a heading level	Flow
<header>	Denotes a heading region	Flow
<hgroup>	Hides all but the first of a set of headings from the document outline	Flow
<nav>	Denotes a significant concentration of navigation elements	Flow
<section>	Denotes a significant concept or topic	Flow
<summary>	Denotes a title or description for the content in an enclosing <details> tag	N/A

Creating Tables

The tags in Table 2-5 are used to create tables or show data in a grid.

Table 2-5. *The Table Tags*

Tag	Description	Type
<caption>	Adds a caption to a table	N/A
<col>	Denotes a single column	N/A
<colgroup>	Denotes a group of columns	N/A
<table>	Denotes a table	Flow
<tbody>	Denotes the body of the table	N/A
<td>	Denotes a single table cell	N/A
<tfoot>	Denotes a footer for a table	N/A
<th>	Denotes a single header cell	N/A
<thead>	Denotes a header for the table	N/A
<tr>	Denotes a row of table cells	N/A

Creating Forms

The tags in Table 2-6 are used for creating HTML forms you can use to obtain input from the user.

Table 2-6. *The Form Tags*

Tag	Description	Type
<button>	Denotes a button that will submit or reset the form. This can also be used as a generic button.	Phrasing
<datalist>	Defines a set of suggested values for the user.	Flow
<fieldset>	Denotes a group of forms tags.	Flow
<form>	Denotes an HTML form.	Flow

(continued)

Table 2-6. *(continued)*

Tag	Description	Type
<input>	Denotes a control to gather data from the user.	Phrasing
<keygen>	Generates a public/private key pair.	Phrasing
<label>	Denotes a label for a form element.	Phrasing
<legend>	Denotes a descriptive label for a <fieldset> tag.	N/A
<optgroup>	Denotes a group of related <option> tags.	N/A
<option>	Denotes an option to be presented to the user.	Phrasing
<output>	Denotes a result of a calculation.	Phrasing
<select>	Presents the user with a fixed set of options.	Phrasing
<textarea>	Allows the user to enter multiple lines of text.	Phrasing

Embedding Content Tags

The tags presented in Table 2-7 are used to embed content into an HTML document.

Table 2-7. *The Form Tags*

Tag	Description	Type
<area>	Denotes an area for a client-side image	Phrasing
<audio>	Defines an audio resource	N/A
<canvas>	Provides a dynamic graphics canvas	Phrasing/flow
<embed>	Embeds content in an HTML document using a plugin	Phrasing
<iframe>	Embeds one document in another by creating a browsing context	Phrasing

(continued)

Table 2-7. *(continued)*

Tag	Description	Type
	Embeds an image	Phrasing
<map>	Denotes the definition of a client-side image map	Phrasing/flow
<meter>	Embeds a representation of a numeric value displayed within the range of possible values	Phrasing
<object>	Embeds content in an HTML document and can also be used to create browsing contexts and to create client-side image maps	Phrasing/flow
<param>	Denotes an option to be presented to the user	Phrasing
<progress>	Embeds a representation of progress toward a goal or completion of a task	Phrasing
<source>	Denotes a media resource	N/A
<svg>	Denotes structured vector content	N/A
<track>	Denotes a supplementary media track, such as subtitle	N/A
<video>	Denotes a video resource	Phrasing

CSS Elements

This section presents CSS selectors and properties. I will not go into any depth in explaining how to use these elements; I will just present them and follow up in later chapters with some examples.

CSS selectors are used to group styles and designate which style is to be used for different HTML tags. Properties are really just styles to be applied to an HTML tag.

To understand the terminology used in the tag names of CSS, the following diagrams show how the elements all relate to each other.

Figure 2-1 is a simple model for a tag (or *element* as it is more formally known). Each tag has a number of surrounding areas that help to set off the tag from other tags.

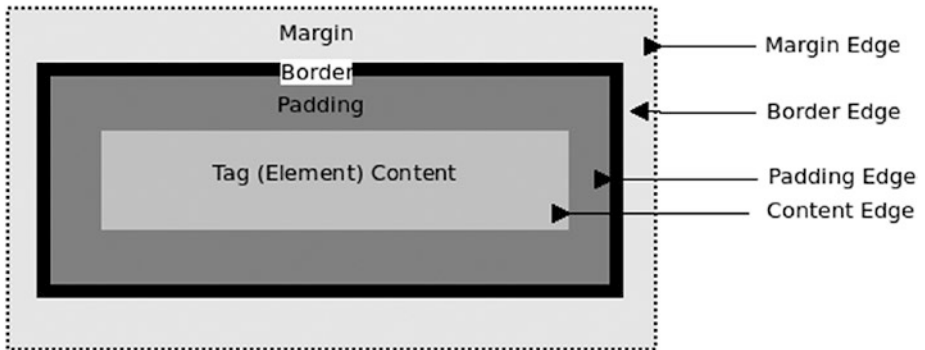


Figure 2-1. *The CSS box model*

Figure 2-2 shows how multiple tags (elements) can be contained in a single box.

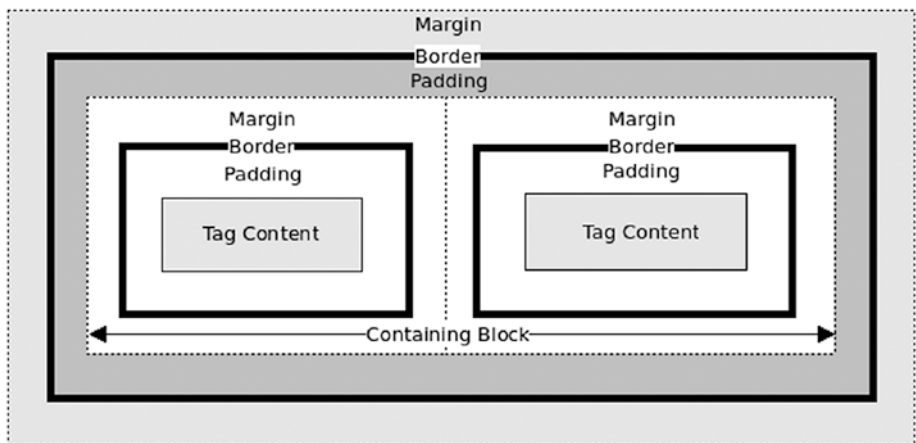


Figure 2-2. *The CSS parent-child box model*

The previous figures show how CSS works and also demonstrate how styles can be cascaded.

CSS Selectors

Table 2-8 lists all the available selectors in CSS.

Table 2-8. The CSS Selectors

Tag	Description
*	Selects all tags.
<type>	Selects tags of the specified type.
.<class>	Selects tags of the specified class.
#<id>	Selects tags with the specified value for the id attribute.
[attr]	Selects tags that define the attribute attr, regardless of the value assigned to the attribute.
[attr="val"]	Selects tags that define attr and whose value for this attribute is val.
[attr^="val"]	Selects tags that define attr and whose value for this attribute starts with the string val.
[attr\$="val"]>	Selects tags that define attr and whose value for this attribute ends with the string val.
[attr*="val"]>	Selects tags that define attr and whose value for this attribute contains the string val.
[attr~="val"]>	Selects tags that define attr and whose value for this attribute contains multiple values, one of which is val.
[attr ="val"]>	Selects tags that define attr and whose value for this attribute a hyphen-separated list of values, the first of which is val.
<selector>, <selector>	Selects the union of the tags matched by each individual selector.

(continued)

Table 2-8. *(continued)*

Tag	Description
<code><selector></code> <code><selector></code>	Selects tags that match the second selector and that are arbitrary descendants of the tags matched by the first selector.
<code><selector> ></code> <code><selector></code>	Selects tags that match the second selector and that are immediate descendants of the tags matched by the first selector.
<code><selector> +</code> <code><selector></code>	Selects tags that match the second selector and that immediately follow a tag that matches the first selector.
<code><selector> ~</code> <code><selector></code>	Selects tags that match the second selector and that follow a tag that matches the first selector.
<code>::first-line</code>	Selects the first line of a block of text.
<code>::first-letter</code>	Selects the first letter of a block of text.
<code>:before</code> <code>:after</code>	Inserts content before or after the selected tags.
<code>:root</code>	Selects the root tag in the document.
<code>:first-child</code>	Selects tags that are the first children of their containing tags.
<code>:last-child</code>	Selects tags that are the last children of their containing tags.
<code>:only-child</code>	Selects tags that are the sole tag of their containing tags.
<code>:only-of-type</code>	Selects tags that are the sole tag of their type defined by their containing tag.
<code>:nth-child(n)</code>	Selects tags that are the <i>n</i> th child of their parent.

(continued)

Table 2-8. *(continued)*

Tag	Description
<code>:nth-last-child(n)</code>	Selects tags that are the <i>n</i> th from the last child of their parent.
<code>:nth-of-type(n)</code>	Selects tags that are the <i>n</i> th from their type defined by their parent.
<code>:nth-last-of-type(n)</code>	Selects tags that are the <i>n</i> th from the last child of their type defined by their parent.
<code>:enabled</code>	Selects tags that are in their enabled state.
<code>:disabled</code>	Selects tags that are in their disabled state.
<code>:checked</code>	Selects tags that are in their checked state.
<code>:default</code>	Selects default tags.
<code>:valid</code>	Selects input tags that are <i>valid</i> or <i>invalid</i> based on input validation.
<code>:invalid</code>	
<code>:in-range</code>	Selects constrained input tags that are within or outside the specified range.
<code>:out-of-range</code>	
<code>:required</code>	Selects input tags based on the presence of the <i>required</i> attribute.
<code>:optional</code>	
<code>:link</code>	Selects link tags.
<code>:visited</code>	Selects link tags the user has visited.
<code>:hover</code>	Selects tags that occupy the position on-screen under the mouse pointer.
<code>:active</code>	Selects tags that are presently activated by the user. This usually means tags that are under the pointer when the mouse button is pressed.

(continued)

Table 2-8. *(continued)*

Tag	Description
:focus	Selects the tag that has the focus.
:not(<selector>)	Negates a selection (for example, selects all tags that are not matched by <selector>).
:empty	Selects tags that contain no child tags.
:lang(<language>)	Selects tags based on the value of the lang attribute.
<target>	Selects the tags referred to by the URL fragment identifier.

Border and Background Properties

Table 2-9 lists all the available border and background properties for all tags.

Table 2-9. *The Border and Background Properties*

Property	Description
background	Shorthand value to set all background values.
background-attachment	Sets the attachment of the background to the tag. This is useful when dealing with tags that have scrolling regions.
background-clip	Selects the area in which the background color image is visible.
background-color	Sets the background color.
background-image	Sets the image for the background.
background-origin	Sets the point at which the background image will be drawn.
background-position	Positions the image in the tag's box.

(continued)

Table 2-9. *(continued)*

Property	Description
background-repeat	Specifies the repeat style for the background image.
background-size	Specifies the size at which the background image will be drawn.
border	Shorthand property to set all border values for all edges.
border-bottom	Shorthand property to set all border values for the bottom edge.
border-bottom-color	Sets the color for the bottom edge border.
border-bottom-left-radius	Sets the radius for a corner. It is used for curved borders.
border-bottom-right-radius	Sets the radius for a corner. It is used for curved borders.
border-bottom-style	Sets the style for the bottom edge border.
border-bottom-width	Sets the width for the bottom edge border.
border-color	Sets the color for border for all the edges.
border-image	Shorthand for image-based borders.
border-image-outset	Specifies the area outside the border box that will be used to display the image.
border-image-repeat	Specifies the repeat style for the border image.
border-image-slice	Specifies the offset for the image slices.
border-image-source	Specifies the source for the border image.
border-image-width	Sets the width of the border image.

(continued)

Table 2-9. *(continued)*

Property	Description
<code>border-left</code>	Shorthand to set the border for the left edge.
<code>border-left-color</code>	Sets the color for the left edge.
<code>border-left-style</code>	Sets the style for the left edge border.
<code>border-left-width</code>	Sets the width for the left edge border.
<code>border-radius</code>	Shorthand for specifying curved edges for a border.
<code>border-right</code>	Shorthand to set the border for the right edge.
<code>border-right-color</code>	Sets the color for the right edge.
<code>border-right-style</code>	Sets the style for the right edge border.
<code>border-right-width</code>	Sets the width for the right edge border.
<code>border-top</code>	Shorthand to set the border for the top edge.
<code>border-top-color</code>	Sets the color for the top edge.
<code>border-top-left-radius</code>	Sets the radius for a corner. It is used for curved borders.
<code>border-top-right-radius</code>	Sets the radius for a corner. It is used for curved .
<code>border-top-style</code>	Sets the style for the top edge border.
<code>border-top-width</code>	Sets the width for the top edge border.
<code>border-width</code>	Sets the width for all borders.
<code>box-shadow</code>	Applies one or more drop shadows.
<code>outline</code>	Shorthand property to set the outline in a single declaration.
<code>outline-color</code>	Sets the color of the outline.

(continued)

Table 2-9. *(continued)*

Property	Description
outline-offset	Sets the offset of the outline.
outline-style	Sets the style of the outline.
outline-width	Sets the width of the outline

Box Model Properties

Table 2-10 lists all the available box properties for all tags.

Table 2-10. *The Box Model Properties*

Property	Description
box-sizing	Sets the box to which the size-related properties apply
clear	Clears one or both edges of a floating tag
display	Specifies the display behavior, or rendering, of an element
float	Specifies how an element should float within a page division
height	Sets the height of the tag's containing area
margin	Shorthand property to set the margin for all four edges
margin-bottom	Sets the margin for the bottom edge of the margin box
margin-left	Sets the margin for the left edge of the margin box
margin-right	Sets the margin for the right edge of the margin box
margin-top	Sets the margin for the top edge of the margin box
max-height	Sets the maximum height for the tag
max-width	Sets the maximum width for the tag

(continued)

Table 2-10. *(continued)*

Property	Description
<code>min-height</code>	Sets the minimum height for the tag
<code>min-width</code>	Sets the minimum width for the tag
<code>overflow</code>	Shorthand property to set the overflow style for both axes
<code>overflow-x</code>	Sets the style for handling overflowing content on the x-axes
<code>overflow-y</code>	Sets the style for handling overflowing content on the y-axes
<code>padding</code>	Shorthand property to set the padding for all four edges
<code>padding-bottom</code>	Sets the padding for the bottom edge
<code>padding-left</code>	Sets the padding for the left edge
<code>padding-right</code>	Sets the padding for the right edge
<code>padding-top</code>	Sets the padding for the top edge
<code>visibility</code>	Sets the visibility of a tag
<code>width</code>	Sets the width of a tag

Text Properties

Table 2-11 lists all the available box properties for text tags.

Table 2-11. *The Text Properties*

Property	Description
@font-face	Specifies the web font to use
direction	Specifies the direction of text
font	Shorthand property to set details of the font in a single declaration
font-family	Specifies the list of font families to be used, in order of preference
font-size	Specifies the size of the font
font-variant	Specifies if the font should be displayed in small caps form
font-weight	Specifies the weight (boldness) of the text
letter-spacing	Specifies the space between letters
line-height	Specifies the height of the line of text
text-align	Specifies the alignment of text
text-decoration	Specifies the decoration of text
text-indent	Specifies the indentation of text
text-justify	Specifies the justification of text
text-shadow	Specifies a drop shadow for a block of text
text-transform	Applies a transformation to a block of text
vertical-align	Aligns the text vertically within the line-height
word-spacing	Specifies the spacing between words

Transition, Animation, and Transform Properties

Table 2-12 lists all the available transition, animation, and transform properties for tags that support them.

Table 2-12. *The Transition, Animation, and Transform Properties*

Property	Description
@keyframes	Specifies the animation code.
animation	Shorthand property for animations.
animation-delay	Specifies a delay before an animation starts.
animation-direction	Specifies how alternate repeats of an animation are performed.
animation-duration	Specifies the duration of an animation.
animation-iteration-count	Specifies the number of times an animation will be repeated.
animation-name	Specifies the name of the set of key frames that will be used for an animation.
animation-play-state	Specifies whether the animation is playing or is paused.
animation-timing-function	Specifies the function used to calculate property values between key frames in an animation.
transform	Specifies the property applies a 2D or 3D transformation to an element. This property allows you to rotate, scale, move, skew, etc., elements.
transform-origin	Specifies an origin for which a transform will be applied.
transition	Shorthand property for transitions.
transition-delay	Specifies a delay before the transition starts.
transition-duration	Specifies the duration of a transition.
transition-property	Specifies one or more properties that will be transitioned.
transition-timing-function	Specifies the function used to calculate intermediate property values during the transition.

Other Properties

Table 2-13 lists properties that do not fit neatly into any of the other categories.

Table 2-13. *The Other Properties*

Property	Description
border-collapse	Specifies whether table borders should collapse into a single border or be separated as in standard HTML
border-spacing	Specifies the spacing between table cell borders
caption_style	Specifies the placement of a table caption
color	Specifies the foreground color for an element
cursor	Sets the style of the cursor
empty-cells	Specifies how borders are drawn on empty table cells
list-style	Shorthand property to specify a list style
list-style-image	Specifies an image to be used as a list marker
list-style-position	Specifies the position of a list marker relative to a list item
list-style-type	Specifies the type of marker used in a list
opacity	Sets the transparency for an element
table-layout	Specifies Properties how the sizes of a table is determined

Organizing HTML and CSS Documents

In this section, I will discuss how to organize all the pages and style sheets you will soon be creating. This is covered before any examples of HTML and CSS pages are shown so that you will have some idea about how to organize the data that will later be presented to you.

Normally a layout of the directories for your website will look something like Listing 2-1.

Listing 2-1. This Is the Normal Structure for Website Directories

```
www/  
  cgi-bin/  
  docroot/  
    css/  
    html-lib/  
    images/  
    js/  
  wsgi/
```

This structure does one thing for you: it places the `cgi-bin` and `wsgi` directories outside of the `docroot` directory. This has advantages because the contents of the `cgi-bin` and `wsgi` directories should never be displayed to the user as that invites possible tampering with the executable files located in those directories.

The `css` directory is where you will store your style sheets, and the `html-lib` directory is where you will store your HTML fragments. While the `css` directory is straightforward, the `html-lib` directory may not be familiar to you. Later, we will be building HTML pages dynamically from fragments of pages stored in the `html-lib` directory. If this directory gets very large, you may want to subdivide it further so that it is easier to maintain.

The `images` directory is where you will store all the static images that will be displayed on your HTML pages. At this point we do not differentiate between the different types of images (JPEG, PNG, GIF, etc.), but you may want to further subdivide the images into groups within the `images` directory.

The `js` directory is where you keep your shared JavaScript programs. We will not use this directory in this book because we will not have a lot of JavaScript in our examples. Instead, what we need we will code directly in the HTML files.

Now that you have some idea how to organize and store the files we will create in this book, let's move on to some really simple HTML and CSS examples.

Simple HTML and CSS Pages

In this section, you will start coding HTML and CSS pages. We will start with a simple example and continually extend it to become a full-fledged working web page.

To get started, we will code an HTML page and a CSS page that contains nothing but a top header. This should provide you with enough information about how all this hooks together. We will code the HTML page in version 5, which will be used throughout this book. The CSS sheet will be coded in version 4, which has been around for a while.

Listing 2-2 shows the code for the HTML page, and Listing 2-3 shows the CSS page. Figure 2-3 shows the output in the Firefox browser.

Listing 2-2. This Is Our First Example HTML Page

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example1-1</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
  <link rel="stylesheet" type="text/css" href="/css/
    example2-1.css" />
</head>
```

```

<body>
  <!--Begin Header-->
  <div id="header">
    <!-- Begin Picture Column -->
    <div id="header-left">
      <a href="/"></a>
      <br />image by W.David Ashley
    </div>
    <!-- Begin Title Column -->
    <div id="header-right"><h1>Example1-1</h1>
    </div>
  </div>
</body>
</html>

```

Listing 2-3. This Is Our First Example CSS Page

```

/* ++++++ start global general styles ++++++ */
html, body {
  margin: 0px;
  padding: 0px;
  font: 10pt arial, helvetica, sans-serif;
  background: #ffffff;
  color: #000000;
}
/* ++++++ end global general styles ++++++ */
/* ++++++ start global structure styles ++++++ */

```

```
#body {
    position: relative;
    text-align: left;
}

#header {
    position: relative;
    padding-left: 5px;
    padding-top: 5px;
    padding-right: 5px;
    padding-bottom: 0px;
    left: 0px;
    top: 0px;
    width: *;
    height: 165px;
}

#header-left {
    width: 200px;
    height: 160px;
    float: left;
    vertical-align: middle;
    text-align: center;
    font-size: 8px;
    border: 1px solid #ccc;
    padding: 5px 10px 5px 10px;
}

#header-right {
    width: *;
    height: 160px;
    line-height: 90px;
    vertical-align: middle;
```

```
font-size: 22px;
border: 1px solid #ccc;
padding: 5px 10px 5px 10px;
overflow-x: hidden;
}
```

```
/* ++++++ end global structure styles ++++++*/
```

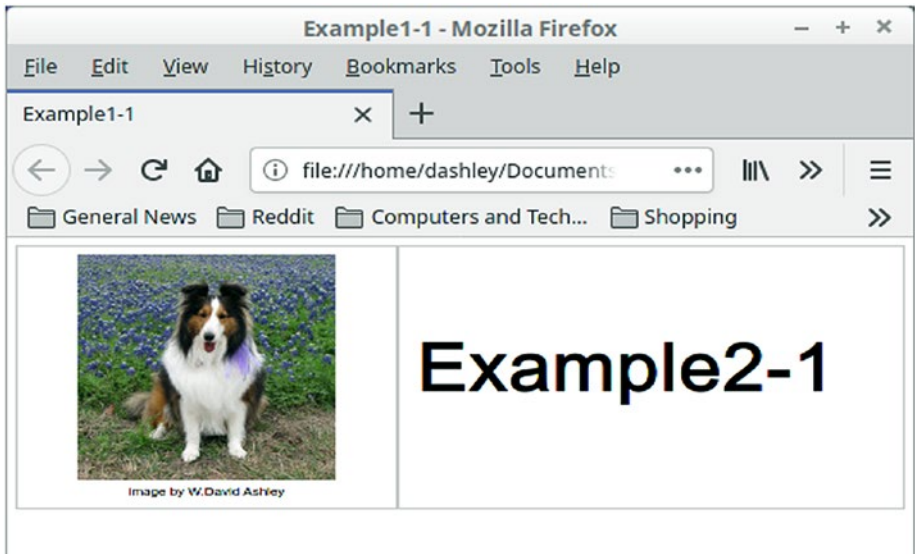


Figure 2-3. *The HTML page displayed in Firefox*

This is a simple example, but let's go over the HTML page and the CSS sheet separately.

The HTML page starts on line 1 by declaring this is an HTML version 5.0 page. The next line starts the HTML block.

The `<head>` tag contains the `<title>` tag, which is displayed in the top bar of the Firefox window. It also contains a `<meta>` tag, which describes the HTML page and some defaults for the browser to use. The next tag is the `<link>` tag, which points to the location of our CSS style sheet.

Next comes the actual body of the page contained within the `<body>` tag. This tag has an `id` of `body`, which corresponds to the `#body` element on the CSS document. The `#body` element specifies the default styling for everything between the `<body>` and `</body>` tags. The style has two specifications: one for the default positions of all the containing blocks (which is `relative` in this case) and one for the default text alignment (which is `left` in this case). This means every block will have these two settings by default unless it is specifically overridden by the contained block specification.

The first block, or `<div id="header">`, within the `<body>` tag is a simple container for two other `<div>` tags. This style for this tag points to the `#header` style and has some simple but needed specifications. The position is `relative`, which means that this container is relative to all other containers at the same level (direct children of the `<body>` level). Padding is specified for the top, bottom, left, and right sides of the elements. The relative origin for the element is specified as `0px` for the upper left. The height is set at `165px`, and the width is the entire width of its container.

The next `<div id="header-left">` within the `<div id="header">` contains an image and some text. We specify a specific width and height for the container so that we can contain the image and the text and match the height of the next container. We also `float` the container to the left to ensure its horizontal position. Next, we center the image and text within the container as well as vertically within the container. We also specify some padding and the font size for all text. Last, we specify a border for the container. Normally we would exclude this item, but we include it here to demonstrate how our containers align.

The last `<div id="header-right">` container will display a title for the page. The `#header-right` style specifies that the width should take up all the remaining horizontal space within the `<div id="header">`. The height of the container should match the height specified in the `<div id="header-left">` so that both containers have the same height. The `line-height` is adjusted to that the text is centered vertically. The

font-size is enlarged so that it is visually pleasing. If the text overflows on the right side, the overflow-x specification simply hides the overflow instead of forcing the text to wrap. We add a little padding just to make sure we have a pleasing look. Lastly, we have our optional border specification just to demonstrate that all the containers line up properly.

At the top of the CSS page are some global values that can be used throughout the rest of the CSS page. These are not likely to change in subsequent parts of the CSS so are placed at the top of the CSS page so they can be used by all of the other styles.

As we can see, we have a nice header that can be reused for many of our documents. We will demonstrate how this is done later in the book.

This simple example demonstrates many of the concepts we will continue to make use of throughout this book.

A Complete HTML and CSS Page

In this section, we will show a complete example HTML and CSS page by extending the short example shown earlier. Listing 2-4 and Listing 2-5 show the complete HTML and CSS pages. Figure 2-4 shows the result.

Listing 2-4. This Is Our Complete Example HTML Page

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example2-2</title>
  <meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
  <link rel="stylesheet" type="text/css" href="/css/
  example2-2.css" />
</head>
<body id="body">
```

```

<!--Begin Header-->
<div id="header">
    <!-- Begin Picture Column -->
    <div id="header-left">
        <a href="/"></a>
        <br />image by W.David Ashley
    </div>
    <!-- Begin Title Column -->
    <div id="header-right"><h1>Example2-2</h1>
    </div>
</div>

<!--Begin Page Menu-->
<div id="pagemenu">
    <!-- Begin Left Title Column -->
    <div id="pagemenu-left">Sample Title
    </div>
    <!-- Begin Page Menu Column -->
    <div id="pagemenu-right"><a href="/">&nbsp;&nbsp;&nbsp;Menu1&nbsp;&nbsp;&nbsp;
    </a>&nbsp;&nbsp;&nbsp;
        <a href="/">&nbsp;&nbsp;&nbsp;Menu2&nbsp;&nbsp;&nbsp;</a>&nbsp;&nbsp;&nbsp;
        <a href="/">&nbsp;&nbsp;&nbsp;Menu3&nbsp;&nbsp;&nbsp;</a>&nbsp;&nbsp;&nbsp;
    </div>
</div>

<!--Begin Content-->
<div id="content">
    <!-- Begin Content Menu Column -->
    <div id="content-left">
        <p>

```

```

        This is just some dummy content for the left
        column
        Normally this would
        be replaced by either hyper links or some other
        selection techniques.
    </p>
</div>
<!-- Begin Content Column -->
<div id="content-right">
    <p>
        This is just some dummy content. This could be
        replaced
        by any combination of text and controls.
        <br/>x<br/>x<br/>x<br/>x<br/>x<br/>x<br/>x<br/>x
        <br/>x<br/>x<br/>x
    </p>
</div>
</div>

<!-- Begin Footer -->
<div id="footer">
    <div id="footer-data">
        
        <br />&#169; Copyright 2010-2020
        W. David Ashley. All rights reserved.
    </div>
</div>
</body>
</html>

```

Listing 2-5. This Is Our Complete Example CSS Page

```

/* ++++++ start global general styles ++++++*/
html, body {
    margin: 0px;
    padding: 0px;
    font: 10pt arial, helvetica, sans-serif;
    background: #ffffff;
    color: #000000;
}
/* ++++++ end global general styles ++++++*/
/* ++++++ start global structure styles ++++++*/

#body {
    position: relative;
    text-align: left;
}

#header {
    position: relative;
    padding-left: 5px;
    padding-top: 5px;
    padding-right: 5px;
    padding-bottom: 0px;
    left: 0px;
    top: 0px;
    width: *;
    height: 165px;
}

```

```
#header-left {  
    width: 200px;  
    height: 155px;  
    float: left;  
    vertical-align: middle;  
    text-align: center;  
    font-size: 8px;  
    border: 1px solid #ccc;  
    padding: 5px 10px 5px 10px;  
    background-color: #F9DAA5;  
}
```

```
#header-right {  
    width: *;  
    height: 155px;  
    line-height: 90px;  
    vertical-align: middle;  
    font-size: 22px;  
    border: 1px solid #ccc;  
    padding: 5px 10px 5px 10px;  
    overflow-x: hidden;  
}
```

```
#pagemenu {  
    float: relative;  
    padding-left: 5px;  
    padding-top: 2px;  
    padding-right: 5px;  
    padding-bottom: 0px;  
    width: *;  
    height: 25px;  
}
```

```
#pagemenu-left {  
    width: 200px;  
    float: left;  
    height: 15px;  
    text-align: center;  
    padding: 5px 10px 5px 10px;  
    background-color: #F9DAA5;  
    border: 1px solid #ccc;  
}
```

```
#pagemenu-right {  
    width: *;  
    height: 15px;  
    overflow-x: hidden;  
    padding: 5px 10px 5px 10px;  
    border: 1px solid #ccc;  
}
```

```
#content {  
    float: relative;  
    padding-left: 5px;  
    padding-top: 2px;  
    padding-right: 5px;  
    padding-bottom: 0px;  
    width: *;  
    height: *;  
}
```

```
#content-left {  
    float: left;  
    width: 200px;  
    min-height: 200px;  
    padding: 5px 10px 5px 10px;
```

```

    background-color: #F9DAA5;
    border: 1px solid #ccc;
}

#content-right {
    width: *;
    height: *;
    padding: 5px 10px 5px 10px;
    overflow-x: hidden;
    border: 1px solid #ccc;
}

#footer {
    float: relative;
    padding-left: 5px;
    padding-top: 2px;
    padding-right: 5px;
    padding-bottom: 0px;
    width: *;
    height: *;
}

#footer-data {
    width: *;
    padding: 5px 10px 5px 10px;
    text-align: center;
    border: 1px solid #ccc;
}

/* ++++++ end global structure styles ++++++*/

```

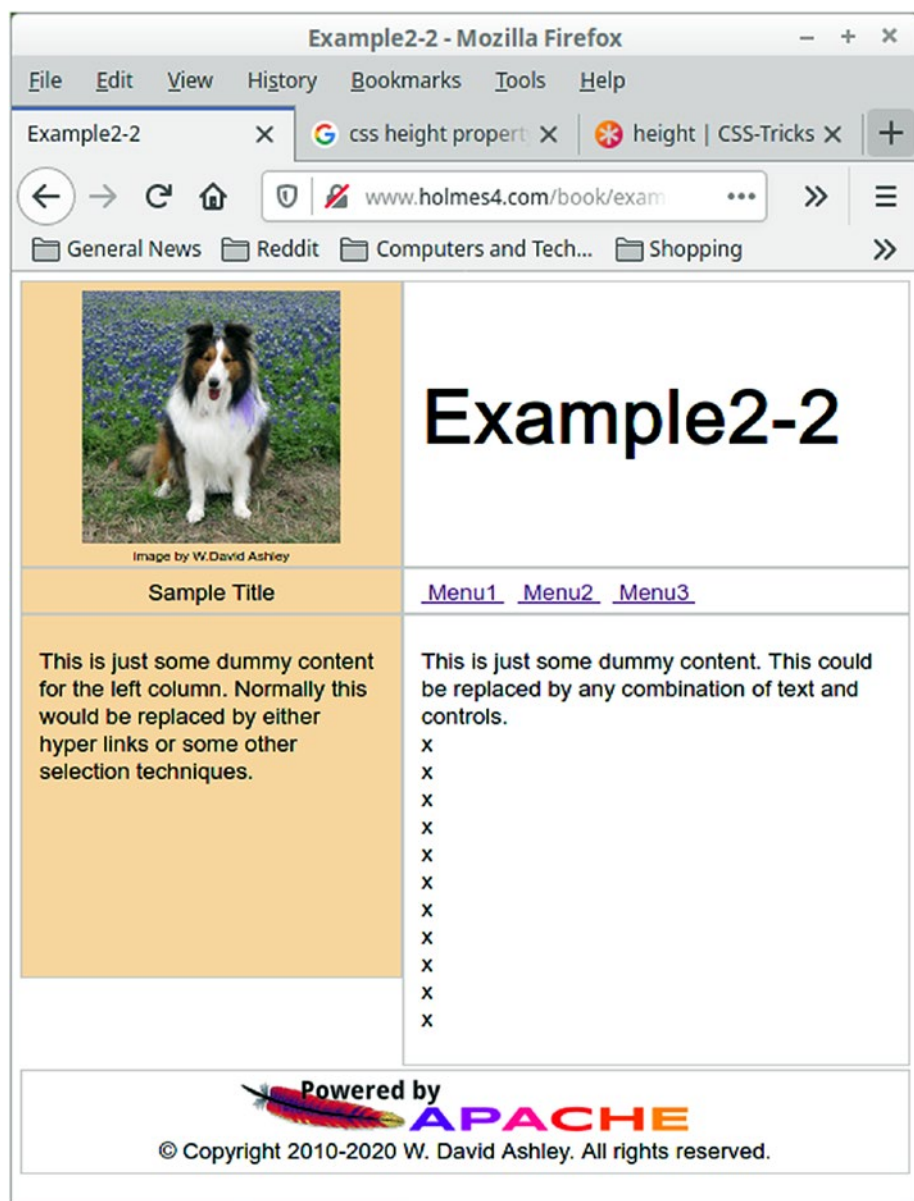


Figure 2-4. The complete HTML page displayed in Firefox

Instead of going over the listings line by line, I will just hit the high points.

The first point that really needs some explanation is in the content-right section of the CSS page. The `overflow-x` element has a property of `hidden`. This is necessary to keep the division from wrapping around to the left and under the content-left section. It keeps all the content in that section in a single straight column. The same is true for the `pagemenu-right` section except that in this case the `overflow-x` property hides any menu entries that overflow to the right instead of wrapping the text. This will happen only when the browser window is sized too small to fit all the menu items.

We also added some color to help the separation of the left content from the right content. This also adds some interest to the page.

Last, we kept the borders for each division so you can see how the divisions relate to each other.

This is how to properly build HTML pages with CSS. If you see any HTML tags or CSS elements you do not understand, you should refer to a more complete reference for HTML/CSS. I provided you with a web page complex enough that we can use it in the following section to explain how to divide an HTML page into parts that can be stored and worked with separately.

Creating a Library of HTML Page Segments

Splitting a page into segments is actually easy if you take the right precautions. You should always try to split an HTML page at major `<div>` segments. This usually has the added benefit of splitting the page at major topic divisions, thus making the segmenting of the page easier from an author's point of view.

The following are some HTML page segments created from Listing 2-4. Listing 2-6 shows the top of the HTML page, or the header as we called it.

Listing 2-6. The HTML Page Header

```

<!--Begin Header-->
<div id="header">
    <!-- Begin Picture Column -->
    <div id="header-left">
        <a href="/"></a>
        <br />image by W.David Ashley
    </div>
    <!-- Begin Title Column -->
    <div id="header-right"><h1>Example2-3</h1>
    </div>
</div>

```

You should note that we did not include the beginning of the HTML file in this segment. That was done on purpose as we may end up using this segment in multiple HTML pages and they may have different content and thus require different headers. Listing 2-7 shows the HTML page menu bar.

Listing 2-7. The HTML Page Menu Bar

```

<!--Begin Page Menu-->
<div id="pagemenu">
    <!-- Begin Left Title Column -->
    <div id="pagemenu-left">Sample Title
    </div>
    <!-- Begin Page Menu Column -->
    <div id="pagemenu-right">
        <a href="/">&nbsp;Menu1&nbsp;</a>&nbsp;
        <a href="/">&nbsp;Menu2&nbsp;</a>&nbsp;
        <a href="/">&nbsp;Menu3&nbsp;</a>&nbsp;
    </div>
</div>

```

Obviously, we want this to be reusable by authors for multiple page designs, so we keep it separate for easier coding. Listing 2-8 shows the major content for the page.

Listing 2-8. The HTML Page Contents

```
<!--Begin Content-->
<div id="content">
  <!-- Begin Content Menu Column -->
  <div id="content-left">
    <p>
      This is just some dummy content for the left
      column. Normally this would
      be replaced by either hyper links or some other
      selection techniques.
    </p>
  </div>
  <!-- Begin Content Column -->
  <div id="content-right">
    <p>
      This is just some dummy content. This could be
      replaced by any combination
      of text and controls.
      <br/>x<br/>x<br/>x<br/>x<br/>x<br/>x<br/>x<br/>x
      <br/>x<br/>x<br/>x
    </p>
  </div>
</div>
```

Both sides of the page are kept together here because if the right-side content changes, then the left side will need to change as well.

Listing 2-9 shows the footer for the HTML page.

Listing 2-9. The HTML Page Footer

```
<!-- Begin Footer -->
<div id="footer">
  <div id="footer-data">
    
    <br />&#169; Copyright 2010-2020
      W. David Ashley. All rights reserved.
  </div>
</div>
```

There is a reason this is broken into a segment. The 2020 date could be programmatically changed easily if the year changed. Otherwise, you would have to change the year manually every time a new year rolled around. We will show you how to do this in the next chapter.

 **Note** It is important to note here that we have been careful to segment each code fragment such that we do not cut it at a point that would also cut the CSS item. If you did this, it could create problems later in the life of the document when it is edited.

Obviously, there are many things that will need to be changed programmatically for all the segments shown previously. This can easily be accomplished via some Python programming. The next chapter will show you how to do that.

Summary

This chapter gave you a look at the HTML tags and CSS items you will be dealing with to create and maintain dynamic pages. You also saw a good directory layout to use on your web server. Last, you saw how to segment your web pages so that they can be utilized and reused by your web page authors.

In the next chapter, I will show you how to build dynamic web pages using the CGI mechanism offered by almost all web servers.

CHAPTER 3

Using CGI and Python

This chapter is all about how to write programs that will be stored in the `cgi-bin` directory. This is a special directory because users can execute programs stored in this directory. You need to be extra careful about the programs you write because they can be abused by the users or anyone else with access to your web server.

Programs stored in the `cgi-bin` directory can be of any type available that are executable on the server. This means they can be written in C, Python, Perl, a command-line shell, or any other language supported on the server. In this book, we will be discussing how to write Python scripts that are stored in the `cgi-bin` directory.

Most web servers come with `cgi-bin` installed. You do not need to do anything except place your programs in the directory with the correct permissions. If you have to create the `cgi-bin` subdirectory, please refer to Listing 1-1 for an explanation of what must be configured to make it work properly.

In this chapter, I will show a few examples of `cgi-bin` programs as well as expand on some of the concepts we used in the previous chapter.

Your First `cgi-bin` Program

In this section, I will show you a simple but useful Python `cgi-bin` program. When executed, this program will display all the environment variables passed to the program on a standard web page.

The program, shown in Listing 3-1, illustrates how to write a `cgi-bin` web program. The first thing to know about this program is that all standard output is routed back to the user's web browser. The browser expects it in a specific format, and this program obeys those rules.

The first item sent back specifies the type of content that is being returned. In this case, it is HTML. This is followed by two carriage returns (linefeeds), sending a line of text with the type followed by a blank line.

Next come the contents of the HTML page. In this case, we have supplied an extremely simple HTML page. We supply the web page prefix HTML tags and then set the font for a title.

Then comes the heart of the page. The loop code runs through the environment data and outputs the name and value of each environment variable.

Last, we close the HTML body and the HTML page.

There are some things to note here. You may think the environment is incomplete because it does not contain some expected values. But this output is actually correct. In the case of `cgi-bin` programs, the environment is completely artificial because it is created by the web server and is not inherited from the web server's environment. This is completely normal because it helps to keep private and secure information out of the hands of rogue `cgi-bin` programs.

This program outputs a lot of useful information, and it shows how the environment can be used to write truly portable programs. It is handy to keep this program around when you are trying to make use of information from the environment.

Listing 3-1. A Useful cgi-bin Program

```
#!/usr/bin/python

import os

print("Content-type: text/html\r\n\r\n")
print("<html><body></br>")
print("<font size=+1>Environment</font></br>")
for param in os.environ.keys():
    line = "<b>%s</b>: %s</br>" % (param, os.environ[param])
    print(line)
print("</body></html>")
```

Listing 3-1 outputs the code in Figure 3-1 when run. Note that there is some line wrapping in this figure.

The bold type names are the environment variable names, and their corresponding values are presented after the names. Again, note that some of the values are wrapped so they would fit in the figure.



Figure 3-1. Output of the useful cgi-bin program

Note that all `cgi-bin` programs when run from this directory will have this current directory, `CONTEXT_DOCUMENT_ROOT`, even if you came from a different web page.

The next item is the `DOCUMENT_ROOT` environment variable. This variable points you to the root directory for the website. We will need this variable later when we want to locate the `html_lib` directory so we can find our HTML segments.

Although it is currently blank, the `QUERY_STRING` environment variable is important to CGI programs. It will contain everything after the `REQUEST_URI` from the command line. Many CGI programs use this information as one or more parameters to be passed to the program. In the case of the `envvar.py` program, we did not pass any `QUERY_STRING` information; thus, it is blank.

For those of you working with multiple languages, the `HTTP_ACCEPT_LANGUAGE` environment variable shows what languages the web server will accept. Only the languages listed will be available to be processed by the web server. The `HTTP_ACCEPT_LANGUAGE` directive works in conjunction with the `LanguagePriority` directive to determine what language will be used when Apache needs to create an HTML page to send to the user.

There are still lots of environment variables left, but they are not as important for our purposes as the ones detailed. They may or may not be useful in your environment, but they are always available for every request.

CGI Program Strategy

This section will outline the strategy for our CGI programs. Basically, there are five steps to each CGI program we will develop.

1. Fetch the HTML parts that will be used for your page.
2. Create the template version of the page.

3. Create a Python dictionary with the values to be substituted into the page template.
4. Substitute the dictionary values into the template to create the HTML page.
5. Send the completed HTML page to the user.

While this may look like a long process, it will become easier with every page you create. It will also be easy to maintain the dynamic pages as each part of the page is handled separately by the Python program.

The substitution process is handled by using the Python Template class from the `string` module. Using this class makes creating the content for the page really easy. Each substitutable part of the page will be contained in a Python dictionary, and then that will be substituted into the template to create the completed HTML page in a single step.

Setting Up the `html_lib` Parts

We are going to use the HTML page created in Chapter 2 as the basis for our dynamic CGI page. But we will need to make some adjustments to accommodate the Python substitution process we plan to use. First, we will make some simple changes to the `html_lib` pieces we created earlier. Listing 3-2 shows the HTML for the header, Listing 3-3 shows the HTML for the menu, Listing 3-4 shows the content, and Listing 3-5 shows the HTML for the footer.

Listing 3-2. The Header HTML Part

```
<!--Begin Header-->
<div id="header">
    <!-- Begin Picture Column -->
    <div id="header-left">
```

```

    <a href="/"></a>
    <br />image by W.David Ashley
</div>
<!-- Begin Title Column -->
<div id="header-right"><h1>$title</h1>
</div>
</div>

```

Listing 3-3. The Menu HTML Part

```

<!--Begin Page Menu-->
<div id="pagemenu">
    <!-- Begin Left Title Column -->
    <div id="pagemenu-left">$titleleft
    </div>
    <!-- Begin Page Menu Column -->
    <div id="pagemenu-right">$main_menu
    </div>
</div>

```

Listing 3-4. The Content HTML Part

```

<!--Begin Content-->
<div id="content">
    <!-- Begin Content Menu Column -->
    <div id="content-left">
        <p>
            $content_left
        </p>
    </div>
    <!-- Begin Content Column -->
    <div id="content-right">

```

```

        <p>
            $content_right
        </p>
    </div>
</div>

```

Listing 3-5. The Footer HTML Part

```

<!-- Begin Footer -->
<div id="footer">
    <div id="footer-data">
        
        <br />&#169; Copyright 2010-$year
        W. David Ashley. All rights reserved.
    </div>
</div>

```

The only difference between these files and the ones from Chapter 2 are the \$variable terms in each part. The variables mark the spots in each part where a substitution will take place. The variable name will correspond to an entry in a Python dictionary, which will then be substituted into the template.

This is all we have to do to a part to get it ready to be used by the Python program. These parts are now ready to be used in many kinds of pages that use our standard web page design.

A Portable and Maintainable CGI Program

Listing 3-6 is our first attempt to write a truly portable and efficient CGI program that is easy to create and easy to maintain.

Listing 3-6. Our First CGI Program

```
#!/usr/bin/python3

import os, time, string

page_top = """Content-type:text/html\r\n\r\n
<!DOCTYPE html>
<html>
<head>
    <title>$title</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
</head>
<body>"""

page_bottom = """</body>
</html>"""

# get the document base for the html_lib directory
html_lib_path = os.getenv('DOCUMENT_ROOT') + '/html_lib/'

# get the page header
f = open(html_lib_path + 'example3-1.html', 'r')
page_header = f.read()
f.close()

# get the page menu bar
f = open(html_lib_path + 'example3-2.html', 'r')
page_menu_bar = f.read()
f.close()

# get the content
f = open(html_lib_path + 'example3-3.html', 'r')
```

```

page_content = f.read()
f.close()

# get the footer
f = open(html_lib_path + 'example3-4.html', 'r')
page_footer = f.read()
f.close()

# create the page template
page_template = string.Template(page_top + page_header +
                                page_menu_bar +
                                page_content + page_footer +
                                page_bottom)

# create the substitutable content
pagedict = dict()
pagedict['title'] = 'My Title'
pagedict['titleleft'] = 'My Left Title'
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'
pagedict['content_left'] = 'Line 1<br/>' + \
                           'Line 2<br/>' + \
                           'Line 3<br/>' + \
                           'Line 4<br/>'
pagedict['content_right'] = ""<h2>Welcome</h2><p>Welcome to the
                           <a href="/" target='_top'>
                           Dummy</a>
                           web site.</p>

```

```

<br/>x<br/>x<br/>x<br/>x<br/>x
<br/>x<br/>x
"""

asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

# make the substitutions
html = page_template.substitute(pagedict)

# print the substituted lines
print(html)

```

Listing 3-6 is our first CGI program that encompasses all of our desired designs, and it includes all of the strategies we described earlier. As you can see, the Python script is both short and easily understood.

The script starts by defining the prefix and suffix HTML code. The reason this is done in the script and not in an HTML part file is that you may have changes that need to be made, especially to the prefix data. So, it is better that it be stored here.

Next, we define the string to contain the `html_lib` files.

Now we can fetch each one of the HTML parts and store them temporarily. After fetching the HTML parts and setting the substitutable variables, we concatenate all the pieces together and turn it into a template.

Next, we define all the entities we need to substitute. This will include all of the entities throughout the entire template. If you leave any entities undefined, an error will be thrown by Python.

Finally, we substitute all the entities into an HTML variable. Then we print the variable, which sends it to STDOUT. This in turn is redirected to the TCP/IP port connected to the client browser.

That is all there is to it. Of course, this is just a small example page. Your pages will have much more content to assign to the `content_left`

and `content_right` dictionary entries. There are some different ways to perform this assignment. If the content is static, you may assign all the content in the Python program, or you could store it in an HTML part file and bring it in at runtime. If you content is variable, then you might choose to perform all the assignments in the script or use a mixed approach. All these methods are viable and could be used as your circumstances dictate.

Figure 3-2 shows the output of this Python script in a browser.

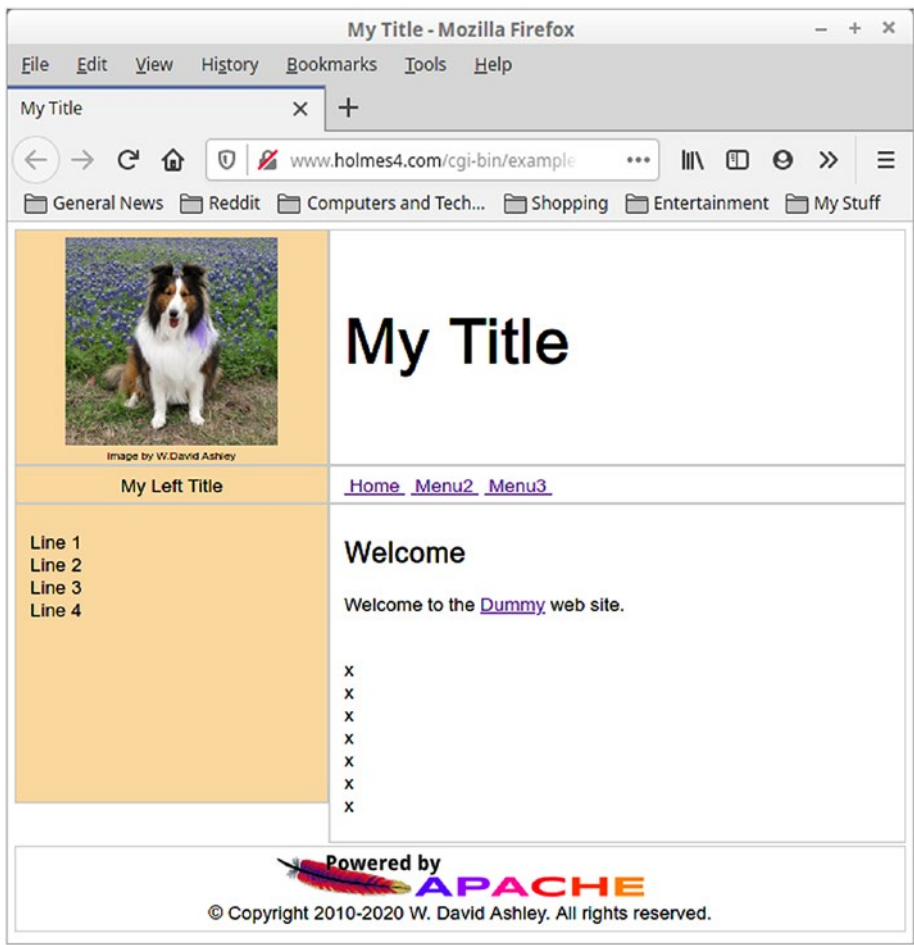


Figure 3-2. Output from Listing 3-6

As you can see, all our substitutions were used for a template to form a full and correct HTML page. The page is still just a short example, but it has all our desired design elements. There are some nice touches that should be noted.

The end date in the copyright notice is completely dynamic. You will never have to edit the page to modify the copyright date as it is updated in the Python script.

There are two titles that appear in three places. The first is the main title, which appears at the top of the browser window and at the top of the HTML page. The second title appears on the left side of the menu bar in the HTML page. This title will probably be different than the main title and different for many types of pages.

Of course, you will have your own menu entries for your website on the page. This is easily changed in the script.

The content in the `content_left` element can be anything that makes sense for the page. Later we will show an example of what can be done with it.

The `content_right` element can have any content that you want, but it should be the main content for the page.

This might be a good time to discuss the picture in the upper left of the page. We have included this one because it is one of my favorites. The dog's name is Dora, and she passed away in late 2016. She was a certified therapy dog and made more than 600 visits to nursing homes, assisted living centers, hospitals, colleges, and special events. She went everywhere I could take her, and she loved people. But in this case, she is just a placeholder for your own logo, picture, or whatever you want to post in this position to represent your site.

At this point, we have reached a junction that will require you to make some decisions that will have long-term impacts on your site. We have four parts of our web page that are really dynamic: the left title, the menu options, the left-side content, and the right-side content. For a simple page, it may be best to provide the content in the Python script. But for other

pages you may want to provide some other alternative skeletons for these areas that are stored in the `html_lib` directory. The point I am making is that the system we have designed is flexible enough to accommodate your choices without forcing you into my preconceived ideas for how to set up your site.

More Partial HTML Skeletons

This section will show you some ideas about how to set up some content HTML skeletons. Many of these skeletons are cooperative; in other words, both content areas are cooperating with each other. For instance, we may have a set of selections in the left content area that are hyperlinks to locations in the right content area. The selections in the left side might also require modifications to the right content area, and this in turn might require you to create some JavaScript to manage the changes dynamically. But let's start with a simple case first.

Hyperlink HTML Skeletons

Hyperlink skeletons are relatively easy to implement and manage. The hyperlinks will appear on the left side, and the destination will appear on the right side. If the content on the right side is large enough, it will cause the page to scroll to that portion of the content.

Our first example will be of some static data stored in an expanded HTML fragment (see Listing 3-7). We will be making a change to the content area of our web page and a small change to our script to bring in that update page.

Listing 3-7. A Static HTML Content Fragment

```

<!--Begin Content-->
<div id="content">
  <!-- Begin Content Menu Column -->
  <div id="content-left">
    <p>
      <a href="#begin-print">begin-print</a><br/>
      <a href="#createcustomwidget">create-custom-
        widget</a><br/>
      <a href="#customwidgetapply">custom-widget-apply</
        a><br/>
      <a href="#done">done</a><br/>
      <a href="#drawpage">draw-page</a><br/>
      <a href="#endprint">end-print</a><br/>
      <a href="#paginat">paginat</a><br/>
      <a href="#preview">preview</a><br/>
      <a href="#requestpagesetup">request-page-setup</
        a><br/>
      <a href="#statuschanged">status-changed</a>
    </p>
  </div>
  <!-- Begin Content Column -->
  <div id="content-right">
    <h1>GtkPrintOperations</h1>
    <p>The following table describes the signals used by
      the GtkPrintOperations.</p>
    <table border="1">
      <thead>
        <tr>
          <th>Signal Name</th>
          <th>Additional Parameters</th>

```

```

        <th>Description</th>
    </tr>
</thead>
<tbody>
    <tr>
        <td width="25%"><a id="beginprint"/>
        begin-print</td>
        <td width="25%">GtkPrintContext *context</td>
        <td width="50%">The user just finished
        changing printer settings but
        rendering has not yet
        begun.</td>
    </tr>
    <tr>
        <td width="25%"><a id="createcustomwidget"/>
        create-custom-widget</td>
        <td width="25%">None</td>
        <td width="50%">The dialog was just
        displayed. You can return a
        widget or a container widget
        containing multiple widgets
        from the callback function so
        that it will be added as a
        custom page to the dialog's
        GtkNotebook.
        </td>
    </tr>

```

```

<tr>
  <td width="25%"><a id="customwidgetapply"/>
    custom-widget-apply</td>
  <td width="25%">GtkWidget *widget</td>
  <td width="50%">Right before begin-print is
    emitted, this signal is
      emitted if a custom widget
      was added in the
      create-customwidget signal
      handler.</td>
</tr>
<tr>
  <td width="25%"><a id="done"/>done</td>
  <td width="25%">GtkPrintOperationResult
    result</td>
  <td width="50%">Printing completed, and you
    can now view the result.
      You should use gtk_print_
      operation_get_error() to
      check the error message if
      the result was
      GTK_PRINT_OPERATION_RESULT_
      ERROR.</td>
</tr>
<tr>
  <td width="25%"><a id="drawpage"/>
    draw-page</td>
  <td width="25%">GtkPrintContext *context gint
    page_num</td>

```

```

        <td width="50%">Each page must be converted
        into a Cairo context.
                You can use this callback to
                render a page manually.</td>
</tr>
<tr>
        <td width="25%"><a id="endprint"/>
        end-print</td>
        <td width="25%">GtkPrintContext *context</td>
        <td width="50%">All of the pages were
        rendered.</td>
</tr>
<tr>
        <td width="25%"><a id="paginat"/>paginat</td>
        <td width="25%">GtkPrintContext *context</td>
        <td width="50%">This signal is emitted after
        begin-print but before page
                rendering begins. It will
                continue to be emitted until
                FALSE is returned or until it
                is not handled. This allows
                you
                to split the document into
                pages in steps so that the
                user interface is not
                noticeably blocked.</td>
</tr>

```

```

<tr>
  <td width="25%"><a id="preview"/>preview</td>
  <td width="25%">GtkPrintOperationPreview
    *preview
      GtkPrintContext *context
      GtkWindow *parent</td>
  <td width="50%">The user requested a preview
    of the document from
      the main printing dialog.
      This signal allows you to
      create
      your own preview dialog. If
      this signal is not handled,
      the
      default handler will be used.
      The callback function returns
      TRUE if you are handling the
      print preview.</td>
</tr>
<tr>
  <td width="25%"><a id="requestpagesetup"/>
    request-page-setup</td>
  <td width="25%">GtkPrintContext *context gint
    page_num GtkPageSetup
      *setup</td>
  <td width="50%">This signal is emitted for
    every page, which gives you one last
      chance to edit the setup of
      a page before it is printed.
      Any

```

```

                                changes will be applied to
                                only the current page!</td>
</tr>
<tr>
    <td width="25%"><a id="statuschanged"/>
    status-changed</td>
    <td width="25%">None</td>
    <td width="50%">Possible values are defined
    by the GtkPrintStatus
                                enumeration, and the current
                                value can be retrieved with
                                gtk_print_operation_get_
                                status().</td>
</tr>
</tbody>
</table>
</div>
</div>

```

Listing 3-8 shows how we have made changes to the content area to enable our Python script to show the data. This makes the static data easy to edit as it ensures that the rest of the web page remains untouched.

Listing 3-8. Python Script to Display Listing 3-7

```
#!/usr/bin/python3

import os, time, string

page_top = """Content-type:text/html\r\n\r\n
<!DOCTYPE html>
<html>
<head>
    <title>$title</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
</head>
<body>"""

page_bottom = """</body>
</html>"""

# get the document base for the html_lib directory
html_lib_path = os.getenv('DOCUMENT_ROOT') + '/html_lib/'

# get the page header
f = open(html_lib_path + 'example3-1.html', 'r')
page_header = f.read()
f.close()

# get the page menu bar
f = open(html_lib_path + 'example3-2.html', 'r')
page_menu_bar = f.read()
f.close()
```

```

# get the content
f = open(html_lib_path + 'example3-6.html', 'r')
page_content = f.read()
f.close()

# get the footer
f = open(html_lib_path + 'example3-4.html', 'r')
page_footer = f.read()
f.close()

# create the page template
page_template = string.Template(page_top + page_header +
                                page_menu_bar +
                                page_content + page_footer +
                                page_bottom)

# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Example3-7'
pagedict['titleleft'] = 'Signals'
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'

asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

```

```
# make the substitutions
html = page_template.substitute(pagedict)

# print the substituted lines
print(html)
```

Listing 3-8 is almost the same as our previous code but with two changes. The main title has been changed, and the content HTML fragment is different. Those are the only changes. An alternative to this code would be to pass the content area fragment file as a parameter to the code instead of hard-coding it.

Figure 3-3 shows the output from the code in Listing 3-8.

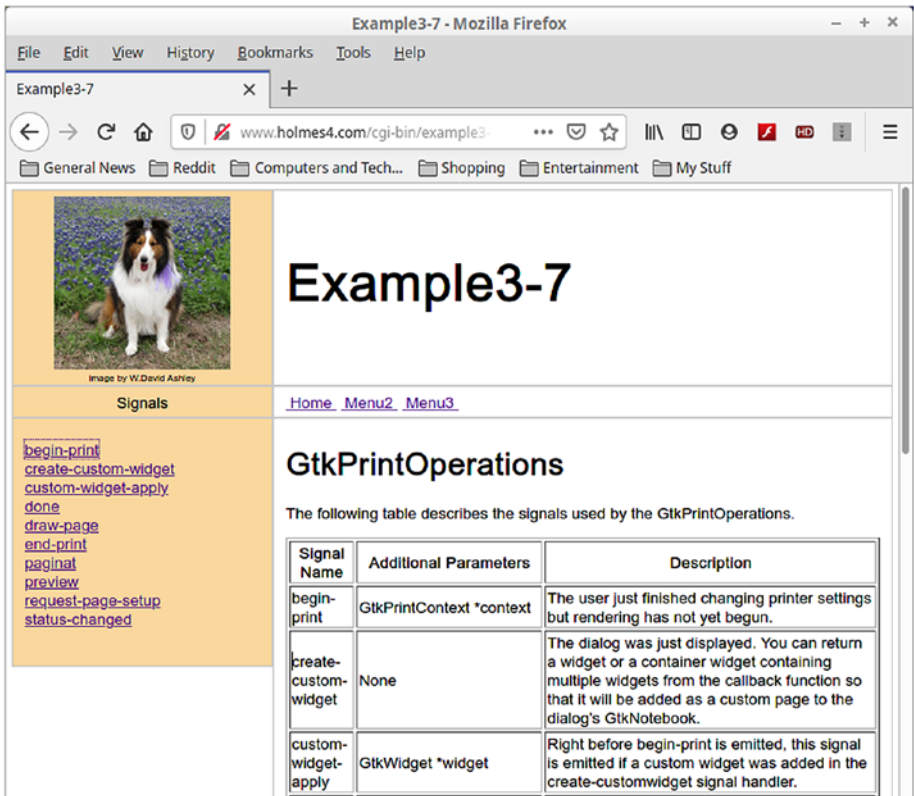


Figure 3-3. Output from Listing 3-8

You can now see how small changes can have a major impact on how our skeletons display a variety of data with relatively little work on the part of the administrator/developer. Writers can now concentrate on writing the output and not on how to display that data.

A More Complicated Hyperlink Example

The following example is a little more complicated than the previous one. It consists of a select HTML element (or list box if you prefer) and a picture element. When you select an entry in the select element, it will display the corresponding picture on the right side on the content area.

The example does not contain any new skeleton file from the `html_lib` directory. Instead, all the work is accomplished in the Python script. See Listing 3-9.

Listing 3-9. An HTML Select Hyperlink Example

```
#!/usr/bin/python3

import os, time, string

page_top = """Content-type:text/html\r\n\r\n
<!DOCTYPE html>
<html>
<head>
    <title>$title</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
    <script type="text/javascript">
        function displayImage(elem) {
            var image = document.getElementById("main-picture");
```

```

        image.src = elem.value;
    }
</script>
</head>
<body>"""

page_bottom = """</body>
</html>"""

# get the document base for the html_lib directory
docroot = os.getenv('DOCUMENT_ROOT')
html_lib_path = docroot + '/html_lib/'

# get the page header
f = open(html_lib_path + 'example3-1.html', 'r')
page_header = f.read()
f.close()

# get the page menu bar
f = open(html_lib_path + 'example3-2.html', 'r')
page_menu_bar = f.read()
f.close()

# get the content
page_content = """
    <!--Begin Content-->
    <div id="content">
        <!-- Begin Content Menu Column -->
        <div id="content-left">
            <select id="pic-list" style="margin-left:10px;
                width:160px"
                                size="20" onchange="displayImage(
                                    this);">
<!--- **** List of picture files goes here **** -->

```

```

$pictures
<!--- **** End list of picture files **** -->
    </select>
</div>
<!-- Begin Content Column -->
<div id="content-right">
<p>It could take some time for the image to load,
please be
    patient.</p>
<img id="main-picture" width="400" border="0" />
</div>
</div>"""

# get the footer
f = open(html_lib_path + 'example3-4.html', 'r')
page_footer = f.read()
f.close()

# create the page template
page_template = string.Template(page_top + page_header +
                                page_menu_bar +
                                page_content + page_footer +
                                page_bottom)

# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Pictures'
pagedict['titleleft'] = 'Pictures'
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'

```

```

# get the list of all files in the picture path
picpath = docroot + '/book/pictures'
files = os.listdir(picpath)
files.sort()
# if the file is a picture then include it in the list,
otherwise skip it
piclist = ''
for fname in files:
    if fname.find('.jpg') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    if fname.find('.JPG') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    if fname.find('.png') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    if fname.find('.PNG') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    continue

```

```

    if fname.find('.gif') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    if fname.find('.GIF') >= 0:
        piclist = piclist + '
                                <option value="" +
                                '/book/pictures/' +
                                fname + '>' + fname +
                                '</option>\n'

        continue
    pagedict['pictures'] = piclist
    asctime = time.asctime()
    parts = asctime.split()
    pagedict['year'] = parts[4]

    # make the substitutions
    html = page_template.substitute(pagedict)

    # print the substituted lines
    print(html)

```

There are a few things to point out in this example. First, a JavaScript function has been added near the top of the HTML part of the listing. This function is called when a new selection has been made from the select element (the list box). The onchange option on the select element calls this function when a new selection has been made on the select element.

After that, everything is pretty standard until we get to the content area further down in the listing. Rather than define the content area in a code fragment, we have decided to define it in the Python script. The decision was purely arbitrary because this is probably the only time we will use this combination of elements in the content area.

Finally, we have to get the list of pictures and add that to the HTML code so the select element is loaded when the HTML is displayed to the user. The picture-loading code will load all kinds of picture files because they are all supported by the select element.

We now have an HTML page that is easy to use and can contain a large selection of pictures. Figure 3-4 shows the output of the code.

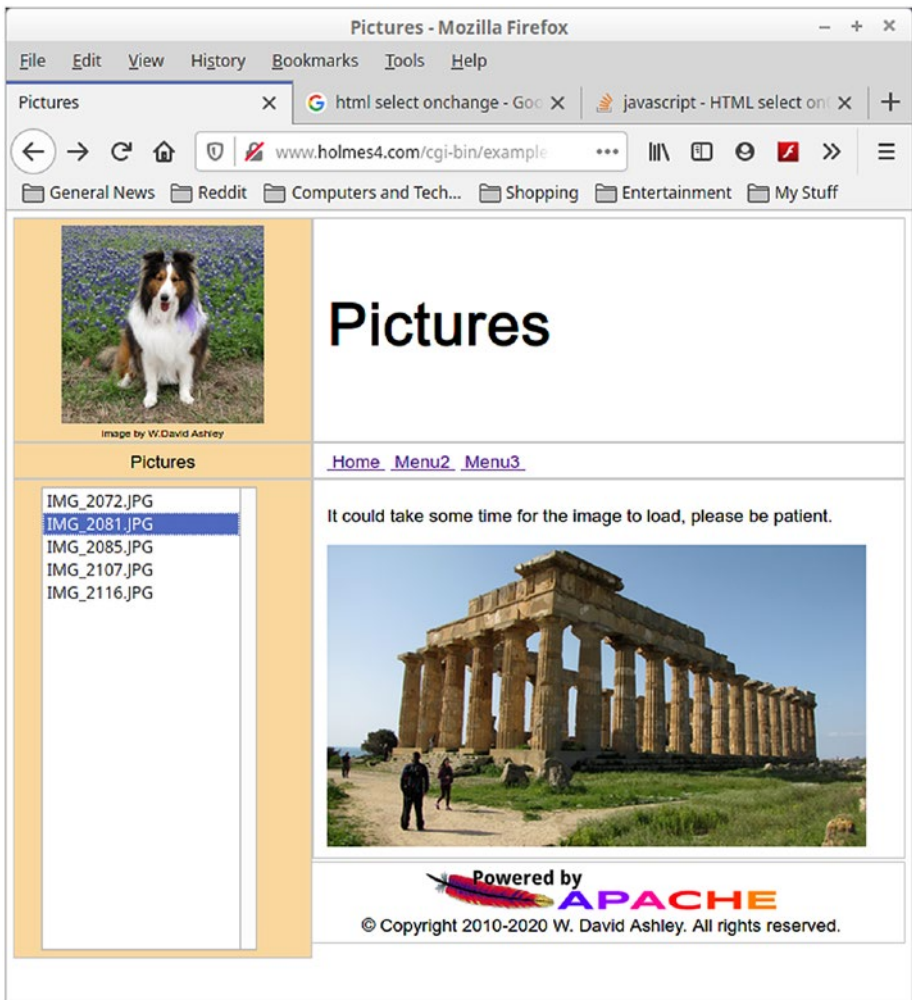


Figure 3-4. Output from Listing 3-9

If you study the code for this example, you will discover one of the weaknesses of CGI programs. Look closely and you will see that we hard-coded the directory where the pictures are stored into the Python script. This is because we can never use a CGI program to “drill down” into the file system because a CGI program always has the same current directory. We could pass the directory we want to use to the script, but how would you know where you came from or where the next location is? You would have to jump through some unusual hoops to keep track of all this. The next chapter will show you a solution to this problem.

Calling and Passing Data to a CGI Program

Up until now, we have just shown some simple CGI programs without showing you how to call or pass data to a CGI program. Calling a CGI program is simple and only involves coding a proper URL. Listing 3-10 shows an example.

Listing 3-10. Calling a CGI Program

```
http://www.yourserver.com/cgi-bin/your_python_pgm
```

The URL in Listing 3-10 has the standard parts. The first part (http) specifies the protocol to be used. In our example, this is the HTTP protocol, but it could also be the HTTPS protocol (the secure HTTP protocol). Either can be used, and it has no impact on how your Python program will be coded.

The next part is the server name or its IP address. Either can be used, but it is obviously easier to remember the server name as its address may change without notice to the user.

Next is the directory location of your program. Most CGI programs are stored in the `cgi-bin` directory as this is a secure location for CGI programs. They can be in other directories, but these should be considered as insecure locations as they can expose the program’s source code to the user.

Next is the name of your CGI program. In our case, we will always use a Python program, and it will always have a `.py` extension. But you should understand this is not a safe practice. Python programs can be made executable without an extension, which makes them safer. This is considered good practice and is highly recommended for your environment.

CGI programs can be coded in any program language supported by your server. Some popular languages are Python, Perl, Bash, Ruby, and PHP. These are all interpreted languages, but you could also use C or C++. All of these languages use the same calling and data passing mechanism because you are passing data to the HTTP server and not directly to the program itself.

The HTTP server may use two methods of passing data to a program. These are known as the GET and POST methods. We will examine both methods in the following sections.

The GET Method

The GET method involves passing data as an appended set of information to the URL request. The information is appended to the URL request and is separated by the `?` character, as shown in Listing 3-11.

Listing 3-11. An Example GET Method with Parameters

```
http://www.example.com/cgi-bin/hello.py?name1=value1&name2=value2
```

The GET method is the default method of passing parameter information from the browser to the web server, and it also produces a string of text in the browser's location box. This method is insecure, so never use it to pass sensitive data to the server such as a password or other sensitive information. The GET method has a size limitation of 1,024 bytes in the URL request string. The data you send to the server appears in the environment using the `QUERY_STRING`, which we saw earlier. Each name/variable is separated from the next via an ampersand (`&`).

A Simple Example GET Method

Listing 3-12 shows an example of a simple GET method, which will pass two values to the program `example3-9`.

Listing 3-12. Using the GET Method to Pass Data

```
http://www.holmes4.com/cgi-bin/example3-9.py?first_name=David&
lastname=Ashley
```

Listing 3-13 shows the `example 3-9.py` program, which handles the input given by the web browser. Note that we are using the Python `cgi` module, which makes it easy to access the arguments passed by the browser.

Listing 3-13. Python Program to Display GET Method Data

```
#!/usr/bin/python3

import os, time, string
import cgi, cgiib

form = cgi.FieldStorage()

page_top = """Content-type:text/html\r\n\r\n
<!DOCTYPE html>
<html>
<head>
    <title>$title</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
    <script type="text/javascript">
        function displayImage(elem) {
            var image = document.getElementById("main-picture");
```

```

        image.src = elem.value;
    }
</script>
</head>
<body>"""

page_bottom = """</body>
</html>"""

# get the document base for the html_lib directory
docroot = os.getenv('DOCUMENT_ROOT')
html_lib_path = docroot + '/html_lib/'

# get the page header
f = open(html_lib_path + 'example3-1.html', 'r')
page_header = f.read()
f.close()

# get the page menu bar
f = open(html_lib_path + 'example3-2.html', 'r')
page_menu_bar = f.read()
f.close()

# get the content
f = open(html_lib_path + 'example3-3.html', 'r')
page_content = f.read()
f.close()

# get the footer
f = open(html_lib_path + 'example3-4.html', 'r')
page_footer = f.read()
f.close()

# create the page template

```

```

page_template = string.Template(page_top + page_header +
                                page_menu_bar +
                                page_content + page_footer +
                                page_bottom)

# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Hello'
pagedict['titleleft'] = ''
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'
pagedict['content_left'] = ''
# test to see if we actually have any input
if 'first_name' not in form:
    first_name = ''
else:
    first_name = form.getvalue('first_name')
if 'last_name' not in form:
    last_name = ''
else:
    last_name = form.getvalue('last_name')
data = """"<form action="/cgi-bin/example3-9.py" method="get">
First Name: <input type="text" name="first_name"/><br/>
Last Name: <input type="text" name="last_name"/><br/><br/>
<input type="Submit" value="Submit"/>
</form><br/><br/>
<h2>Hello """"

```

```

pagedict['content_right'] = data + first_name + ' ' + last_name
+ '</h2>'
asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

# make the substitutions
html = page_template.substitute(pagedict)

# print the substituted lines
print(html)

```

The first thing to note about this script is the new `import` statement to include the `cgi` and `cgitb` modules. We will use these to help us access the GET arguments passed to the script.

Next is the statement to access the GET arguments via the `cgi.FieldStorage()` method. This will obtain all the GET arguments for us.

The rest of the work is contained in the `content_right` part of the code. We then test each argument to see whether it actually exists in the form variable. If it does not, then we set the corresponding Python variable to a zero-length string. Then we can construct the HTML form as well as display the Python name variables.

Figure 3-5 shows the output of our example.

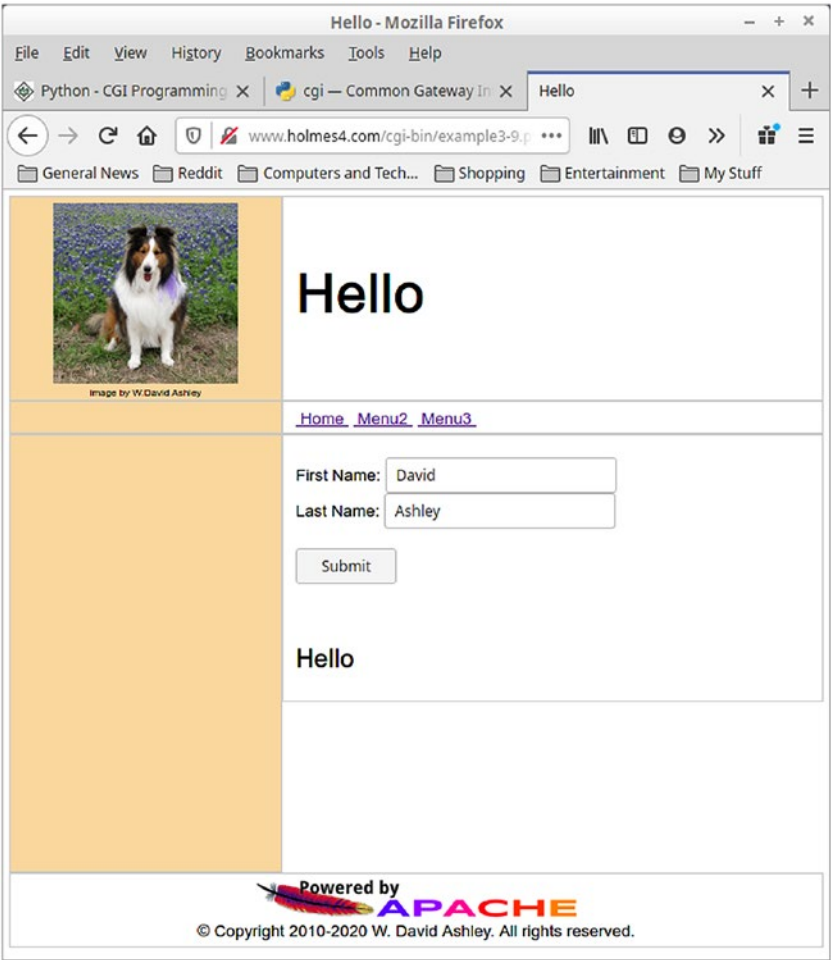


Figure 3-5. Output of the simple GET method example

As you can see, we have successfully displayed the arguments in our example program shown earlier. The main point here is that it does not matter where the input comes from (the command line or the HTML form); the first and last names will always be displayed if it was not a zero-length string.

★ **Important** All parameters and arguments in CGI programs are passed as textual data. If your Python script needs to use the data in either fixed binary or floating-point format, then you will need to convert it from textual to the binary format you need.

The POST Method

The POST method can be invoked only via an HTML form. It may not be invoked from the command line. This makes the data passed to your Python script much safer since it is encrypted to some extent. This is not to say that you cannot pass data via the GET method to a script that in turn uses the POST method to pass on data to another program. Just remember that sensitive data should not be passed to a CGI program via the GET method.

Luckily, the POST method is used almost exclusively in HTML forms. And in Python it does not matter which method you use. The `cgi` module detects which method is being used and collects and presents the arguments to you using the same method. The Python script in Listing 3-14 shows how to utilize check boxes in an HTML form and collect or display the results of the user's interaction with the script.

Listing 3-14. Using the Post Method with Check Boxes

```
#!/usr/bin/python3

import os, time, string
import cgi, cgitb

form = cgi.FieldStorage()

page_top = """Content-type:text/html\r\n\r\n
<!DOCTYPE html>
```

```

<html>
<head>
    <title>$title</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
    <script type="text/javascript">
        function displayImage(elem) {
            var image = document.getElementById("main-picture");
            image.src = elem.value;
        }
    </script>
</head>
<body>"""

page_bottom = """</body>
</html>"""

# get the document base for the html_lib directory
docroot = os.getenv('DOCUMENT_ROOT')
html_lib_path = docroot + '/html_lib/'

# get the page header
f = open(html_lib_path + 'example3-1.html', 'r')
page_header = f.read()
f.close()

# get the page menu bar
f = open(html_lib_path + 'example3-2.html', 'r')
page_menu_bar = f.read()
f.close()

```

```

# get the content
f = open(html_lib_path + 'example3-3.html', 'r')
page_content = f.read()
f.close()

# get the footer
f = open(html_lib_path + 'example3-4.html', 'r')
page_footer = f.read()
f.close()

# create the page template
page_template = string.Template(page_top + page_header +
                                page_menu_bar +
                                page_content + page_footer +
                                page_bottom)

# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Example3-10'
pagedict['titleleft'] = ''
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'
pagedict['content_left'] = ''
# test to see if we actually have any input
if 'science' not in form:
    science_flag = 'OFF'
else:
    if form.getvalue('science'):
        science_flag = 'ON'

```

```

    else:
        science_flag = 'OFF'
if 'english' not in form:
    english_flag = 'OFF'
else:
    if form.getvalue('english'):
        english_flag = 'ON'
    else:
        english_flag = 'OFF'
data = """<form action="/cgi-bin/example3-10.py" method="post">
Science: <input type="checkbox" name="science"
value="on"/><br/>
English: <input type="checkbox" name="english"
value="on"/><br/><br/>
<input type="Submit" value="Select Subjects"/>
</form><br/><br/>
"""

data += '<h2>CheckBox Science is: ' + science_flag + '</h2>'
data += '<h2>CheckBox English is: ' + english_flag + '</h2>'
pagedict['content_right'] = data
asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

# make the substitutions
html = page_template.substitute(pagedict)

# print the substituted lines
print(html)

```

The only changes between this program and the previous program are in the `content_right` part. Here we set up the check boxes and possibly

assign values to them if any arguments are found. If arguments are found, the check boxes are left unchecked.

Once the HTML page is shown, the user may interact with it by choosing to check one or more of the check boxes and then submitting the data. When they submit the form, the HTML page is redisplayed with the results displayed further down the page.

Figure 3-6 shows an example of the output page.

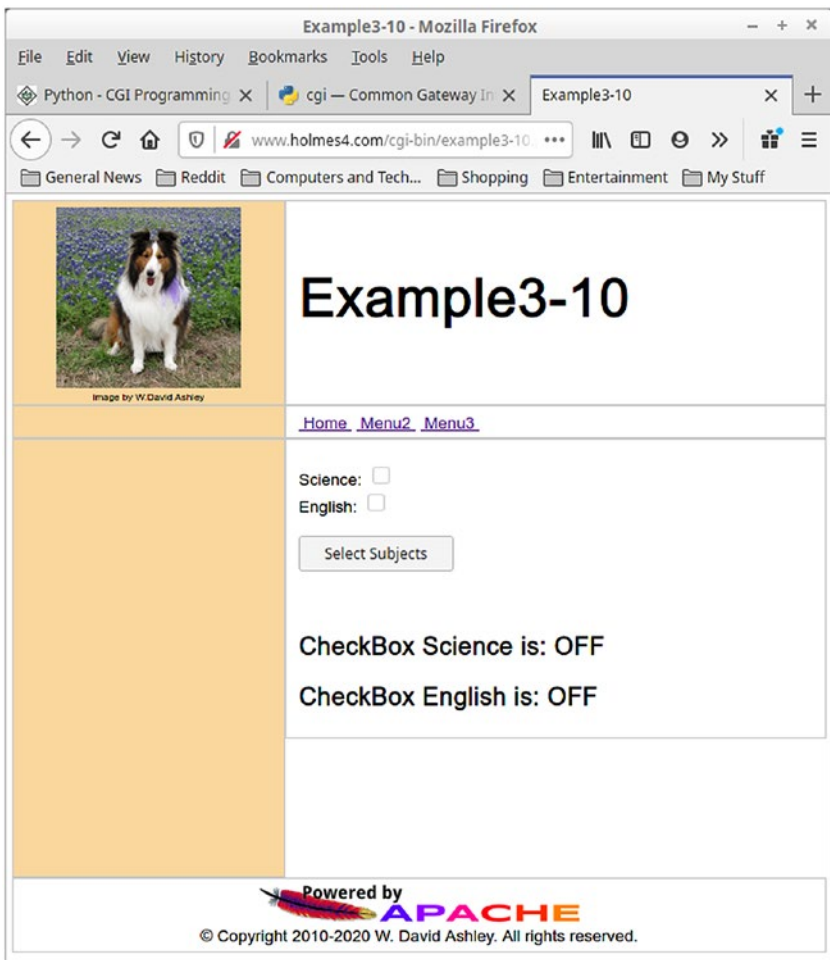


Figure 3-6. Output from Listing 3-14

Figure 3-6 shows the output of our example script along with the check boxes and the output values from the previous submittal. The HTML form collects the form data and sends it back to another invocation of itself.

Another POST Method with HTML Text Data

Listing 3-15 shows how to deal with an HTML textarea. The Python script is similar to the previous examples, with the data assignments for the `content_right` variable being the only real difference. Therefore, Listing 3-15 shows only that area of the program.

Listing 3-15. Using TextArea in a CGI Program

```
# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Example3-11'
pagedict['titleleft'] = ''
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'
pagedict['content_left'] = ''
if form.getvalue('textdata'):
    textdata = form.getvalue('textdata')
else:
    textdata = "Nothing to display!"
data = ""<p>The input text is as follows:</p>""
data += textdata
data += ""<br/><br/><form action="/cgi-bin/example3-11.py"
method="post">
```

```

<textarea name="textdata" cols="40" rows="10">
Type your text here:
</textarea><br/>
<input type="submit" value="Submit"/>
</form>"""
pagedict['content_right'] = data
asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

```

The big difference between this example and the previous one is the assignment to the data variable. Everything needed to get the text data passed to the program appears just in the assignments to the data variable.

One thing to note here is that the `\r\n` characters from the passed data are ignored by the HTML processor. If you need them to be presented to the user, you will need to replace the `\r\n` characters with the HTML `<br/>` tag before assigning the `textdata` variable to the data variable.

Figure 3-7 shows how the data will be presented to the user.

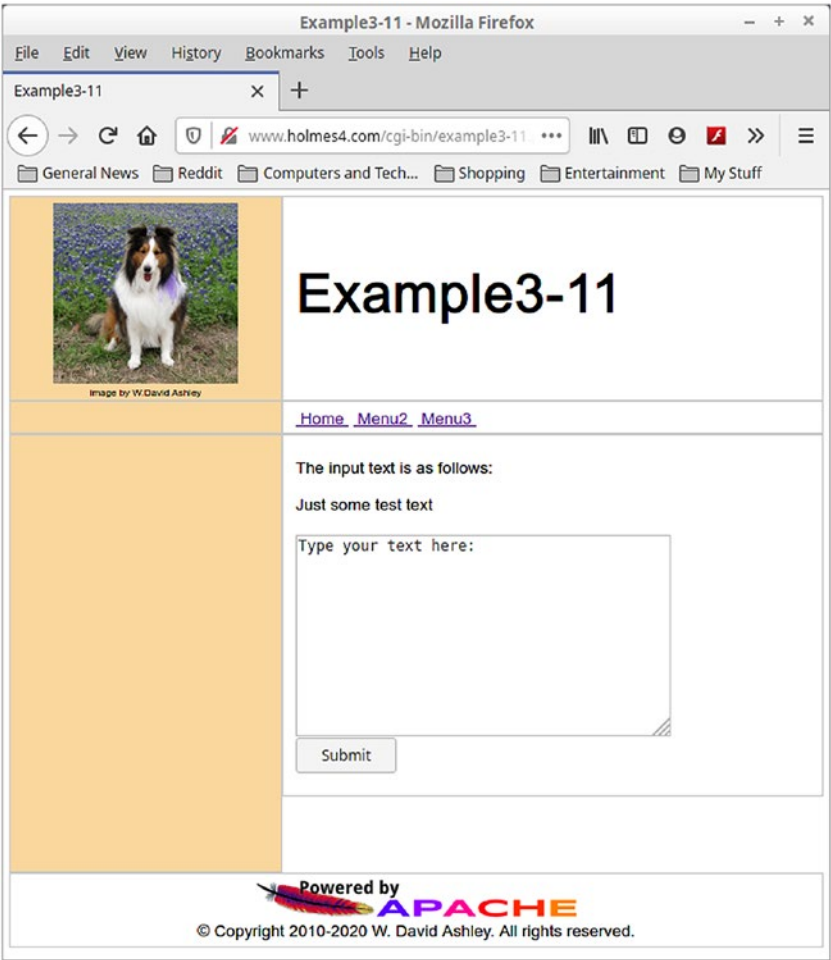


Figure 3-7. The output from Listing 3-15

This previous example is simple but shows how the data is presented to the user after text has been entered and passed to the program via the Submit button. Obviously, the textdata area can hold a great deal of text, which is why we need to make sure we use the POST method to pass the data to the script. The GET method can pass only about 1,024 bytes of data, and the textarea can hold much more than that.

Using POST with a dropdown Box

The next example shows how to utilize an HTML dropdown box. The dropdown box is a compact alternative to the listbox and can even be used as a replacement for a set of radio buttons. Listing 3-16 shows our example Python CGI script for dealing with a dropdown box.

Listing 3-16. Using a dropdown Box in a CGI Program

```
# create the substitutable content
pagedict = dict()
pagedict['title'] = 'Example3-12'
pagedict['titleleft'] = ''
pagedict['main_menu'] = '<a href="/">&nbsp;Home&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu2&nbsp;
                        </a>&nbsp;' + \
                        '<a href="/">&nbsp;Menu3&nbsp;
                        </a>&nbsp;'
pagedict['content_left'] = ''
if form.getvalue('dropdown'):
    subject = form.getvalue('dropdown')
else:
    subject = "Nothing to display!"
data = "<p>The selected subject was: " + subject + "</p>"
data += """"<br/><br/><form action="/cgi-bin/example3-12.py"
method="post">
<select name="dropdown">
<option value="Science" selected>Science</option>
<option value="English">English</option>
</select><br/><br/>
<input type="submit" value="Submit"/>
</form>"""
```

```

pagedict['content_right'] = data
asctime = time.asctime()
parts = asctime.split()
pagedict['year'] = parts[4]

```

The dropdown box selection is easy to obtain using the form. `getvalue()` method. We only need to pass the name of the dropdown box that is assigned to the box. Then we display the selection later in the code.

Figure 3-8 shows the HTML that will be displayed to the user.

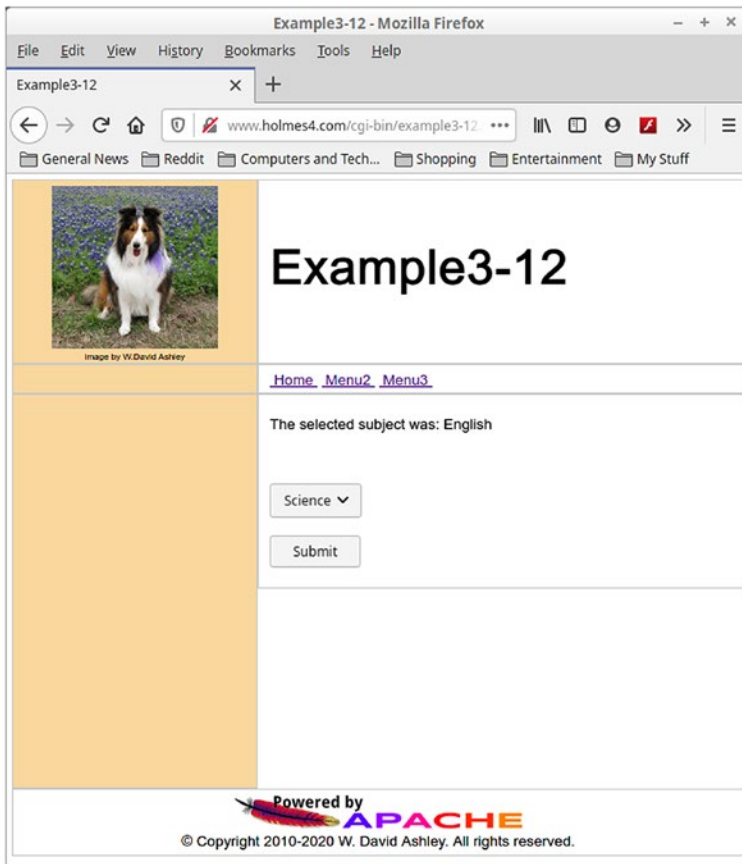


Figure 3-8. The output from the code in Listing 3-16

This simple example is all you need to obtain the user's selection. The dropdown box can hold a lot of possible entries, so it can save a lot of space on your HTML page, thus making it a cleaner design for the user.

Cookies in CGI

The HTTP protocol is what is known as a *stateless* protocol. The server does not maintain any data that connects the user to a set of transactions. Once the CGI program ends a transaction, all memory of what was transferred is lost by the server; in other words, the HTTP server cannot maintain a session with the user between CGI transactions.

To work around this problem, modern web designers use cookies to remember what has transferred between transactions. This provides “pseudo” session state information between the user's browser and the HTTP server. This is especially useful for remembering the user's database login information, file information, items in a cart, or any other information that is required by the transaction.

Sending Cookies from the Server

Cookies always originate from the HTTP server, never from the browser. When the server sends a cookie to the browser, the browser may accept or deny the cookie information. Often this decision is made by the user. If the cookie is accepted, it is saved by the browser. When the user sends information back to the server, the browser includes the previously sent cookie with the new CGI transaction. The server may then utilize the information in any way it chooses.

Cookies are sent by the HTTP server in plain-text form as part of the HTTP header in the HTML returned to the user's browser. It must appear prior to the Content-type header statement. Listing 3-17 shows how to construct a cookie.

- **Path:** The path of the directory or web page that sets the cookie. This can be blank if you want to retrieve the cookie from a directory page or page.
- **Secure:** If this field contains the word `secure`, then the cookie may only be retrieved with a secure server using the HTTPS protocol. If it is blank, no such restriction exists.
- **Name=value:** All cookies have this general form of name-value pairs. You are free to choose your own names provided they do not override the ones listed previously.

All of the standard name-value pairs listed are optional and do not need to appear in a cookie. This is useful when setting temporary cookies.

All `Set-Cookie` statements should appear *before* the `Content-type:text/html/r/n/r/n` line in the HTTP header.

Retrieving Cookies

Cookies may be retrieved easily. They are stored in the CGI environment variable `HTTP_COOKIE`, and they have the following form:

```
Name1=value1;name2=value2;...;namen=valuen;
```

You should note this is the same form as the `QUERY_STRING` environment variable value. The code excerpt in Listing 3-18 should provide an example to help you retrieve cookie information.

Listing 3-18. How to Extract Cookie Key-Value Pairs in Python

```
#!/usr/bin/python3

from os import environ
import cgi, cgitb

userid = None
password = None
if environ.has_key('HTTP_COOKIE'):
    for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
        (key, value) = split(cookie, '=')
        if key == 'UserID':
            userid = value
        if key == 'Password':
            password = value
# the Python variables Userid and password are now set to valid
values or None
```

This code is sufficient to extract any name-value pairs that are known to the server by adding additional tests and assignments.

Summary

This chapter introduced CGI programming using Python. We covered a number of topics in this chapter related to CGI programming including HTML, SSL, and some specific Python techniques to help you in your own coding. The following is a summary of the good and bad of using CGI programming for your project:

- **BAD:** Each time a CGI program is invoked, a new process is spawned by the HTTP server to process the request. When the CGI program finishes, the process

is destroyed. This takes some time and does have an impact on the performance of the HTTP server. However, improvements to the Linux operating system in recent years and newer, faster processors have improved the performance of invoking and tearing down new processes to the point that the impact of a new process for each CGI program is greatly reduced. Nevertheless, this potential problem should be kept in mind.

- *GOOD*: When the HTTP server invokes a new process for a CGI program, it does so with a specialized set of environment variables that hide much of the server's sensitive data.
- *BAD*: As with any programs that interact with users, you should not trust any of the data that comes back from using HTML forms. Users are an inventive lot and will try any way to get the data they want and not necessarily the data you want to provide. Basically, you should always verify, verify, verify all data you receive from a user. If you do not, you will eventually have trouble.
- *BAD*: Do not trust path data provided by a user, especially for file uploads. For that matter, watch out for bad filenames as well.
- *BAD*: Verify all data from the user to make sure it does *not* contain HTML. This type of data can be hard to locate or debug.

- *BAD*: If you are executing external programs in your script, be sure that if information for that program comes from the user, you verify it first.
- *BAD*: CGI script/program permissions under Linux/Unix are also potential problem areas. Be sure that you verify those permission and occasionally ensure they have not been modified.
- *BAD*: Try not to use scripts not authored by you or your organization. These scripts might be performing functions that you do not understand or even know about.

CHAPTER 4

Using SSI and Python

The Server Side Includes (SSI) scripting language triggers further processing on your static web pages. The server parses an HTML page looking for SSI directives. It then processes those directives and creates a new page that in turn is passed back to the user. The modification directives that SSI supplies can do simple text replacement or can call programs to supply the text. The range of possibilities is endless for SSI, and you will see an example of each SSI directive in this chapter.

Getting Started

To use SSI directives in your HTML pages, you must configure SSI in the server configuration file. It is easy to perform this configuration, but it must be placed in the main portion of the config file. There are only two statements to add, as shown in Listing 4-1.

Listing 4-1. SSI Configuration Statements

```
AddHandler server-parsed .shtml  
Options +Includes
```

After adding these statements, you will need to restart the HTTP server to activate them.

Note You may also have to add Includes Indexes FollowSymLinks to make this work with the default settings, as shown here:

```
<Directory "/var/www/html">
    Options Indexes Includes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```

The `.shtml` portion of the `AddHandler` statement specifies the file extension that will be used to tell the server which HTML files need to be parsed for SSI directives. Only files with this extension will be parsed for SSI directives.

If your website is running on a Unix or Linux operating system, there is an alternative available to using a special filename for HTML files that contain SSI directives. The HTTP server directive `XBitHack` allows you to set the owner execute bit on the file to indicate that the server should parse the file for SSI directives. If you choose to use this directive, then your server config file should contain the directives shown in Listing 4-2.

Listing 4-2. SSI Configuration Statements with `XBitHack` Directive

```
AddHandler server-parsed .shtml
Options +Includes
XBitHack on
```

When configured this way, the `.shtml` file extensions will be ignored, and only files with the owner execute bit set will be parsed for SSI directives.

There are additional statements that can be placed in the HTTP server configuration file that will alter the processing of the file by SSI, but we will look at those later in this chapter.

★ **Important** SSI directives placed in the output of CGI programs will *not* be processed. This because no filename is associated with the CGI output, so the server has no way to know whether it needs to be parsed.

SSI directives are really no more than specially formatted HTML comments. This to make them stand out from normal comments and be easily parsed by the server. The general format of an SSI directive follows:

```
<!--#element attribute="value" attribute="value" ... -->
```

The pound sign (#) is what identifies this comment as an SSI directive. If a directive accepts attributes, then the attributes should follow the directive name. There are eight SSI directives plus some miscellaneous ones, and the following sections describe each one.

The config SSI Directive

The config directive controls some aspects of parsing. Three attributes are available, all of which have default values. This directive overrides the default or any previous modification to the attribute. It can appear more than once in your HTML file or not at all. The syntax is as follows:

```
<!--#config errmsg="Your error msg" sizefmt="The format for
display od file sizes"
        timefmt="Format for displaying time values" -->
```

The following are the definitions for each attribute:

errmsg	This is the message sent back to the client when an error occurs during document parsing.
sizefmt	This is the value to use when displaying the size of a file. Valid values are bytes or abbrev for a size in kilobytes or megabytes as appropriate.
timefmt	This value is the same as passed to the C function strftime() library routine.

Listing 4-3 shows a simple HTML page that displays each attribute of the config directive. We deliberately removed all the framing from the example to make it as simple as possible. Obviously, this will work on a more complicated HTML page as well.

Listing 4-3. An Example HTML Page Using the SSI errmsg Directive

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example4-1</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
</head>
<body id="body">
  <h1>Example4-1 - errmsg SSI Directive</h1>
  <p>The following is an example of the <i>config errmsg</i>
    SSI directive. The error message should be
    "This is an example error!".</p>
  <!--#config errmsg="This is an example error!"-->
  <!--#config xerrmsg="This is an example error!"-->
```

```

<br/>
<p>The following is an example of the <i>config sizefmt</i>
    SSI directive. The file size should be presented in either
    kilobytes or megabytes.</p>
<!--#config sizefmt="abbrev"-->
<!--#fsize file="./example4-1.html"-->
<br/>
<p>The following is an example of the <i>config timefmt</i>
    SSI directive. The file time should be presented as
    "%A %Y-%m-%d %H:%M:%S".</p>
<!--#config timefmt="%A %Y-%m-%d %H:%M:%S"-->
<!--#flastmod file="./example4-1.html"-->
</body>
</html>

```

The code in Listing 4-3 will display in the browser as shown in Figure 4-1.

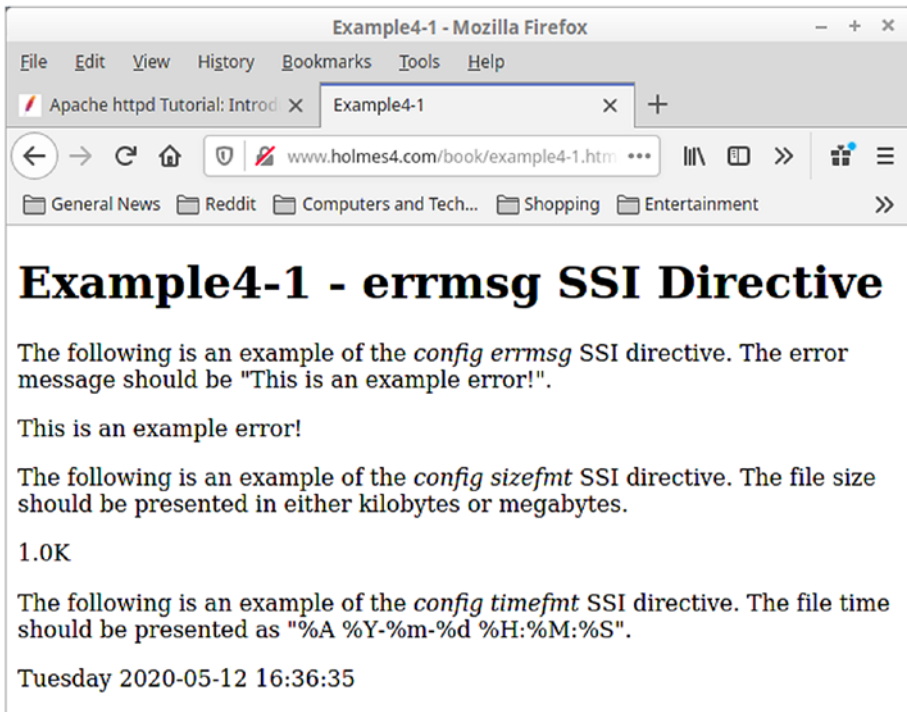


Figure 4-1. Output from Listing 4-3

As you can see, this SSI directive can save you a lot of work updating content in your HTML pages. When the file is modified, the file size and update timestamp can also be updated automatically.

The error message produced can also be modified throughout the document to make it specific to errors in different parts of the document.

The echo SSI Directive

The echo SSI directive prints out the value of an assigned environment variable. A limited number of environment variables can be printed in the document. There is only one attribute available for this directive, and it is documented here:

var This is the name of the environment to be printed. Only the following environment variables may be specified: DATE_GMT, DATE_LOCAL, DOCUMENT_NAME, DOCUMENT_URI, and LAST_MODIFIED.

Listing 4-4 shows an example of each possibility for the echo directive.

Listing 4-4. An Example of the SSI echo Directive

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example4-2</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
</head>
<body id="body">
  <h1>Example4-2 - errmsg SSI Directive</h1>
  <p>The following is an example of the <i>config echo</i>
    SSI directive.</p>
  <p>DATE_GMT = <!--#echo var="DATE_GMT"--></p>
  <p>DATE_LOCAL = <!--#echo var="DATE_LOCAL"--></p>
  <p>DOCUMENT_NAME = <!--#echo var="DOCUMENT_NAME"--></p>
  <p>DOCUMENT_URI = <!--#echo var="DOCUMENT_URI"--></p>
  <p>LAST_MODIFIED = <!--#echo var="LAST_MODIFIED"--></p>
</body>
</html>
```

The code in Listing 4-4 will display in the browser as shown in Figure 4-2.

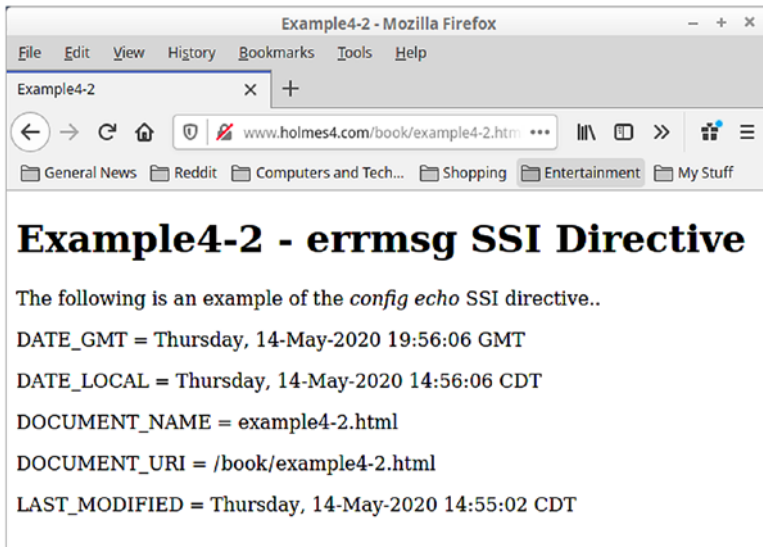


Figure 4-2. Output from Listing 4-4

Although the number of environment variables is restricted, the echo directive is still a convenient directive to use when needed. It can still save time and reduce the number of times an HTML file is updated.

The exec SSI Directive

The exec SSI directive executes either a shell command or a CGI script. A prudent webmaster will put the option IncludesNOEXEC in the server config file to disable this SSI directive. But if available, it allows the output from a shell command, program, or CGI script to be included at that point in the HTML file.

Two attributes are available for the `exec` directive, as documented here:

-
- | | |
|------------------|--|
| <code>cgi</code> | This is a %-encoded URL relative path to the CGI script. If the path does not include a leading slash, it is taken to be relative to the current document. The document is invoked as a CGI script even if the server does not recognize it as a CGI program. All the rules for a directory containing a CGI script must be observed. If <code>suexec</code> is turned on, it will be enforced. The <code>QUERY_STRING</code> and <code>PATH_INFO</code> environment variables passed to the original document will be passed in turn to the specified CGI script as well as the <code>include</code> variables. Note that the <code>include virtual</code> SSI directive should be used in preference to the <code>exec</code> directive if possible. |
| <code>cmd</code> | The server prepends the <code>/bin/sh</code> string to the front of this attribute and then attempts to execute it. The <code>include</code> variables are available to the command, and its output of the program will be included in the HTML file at that point. |
-

 **Important** The documentation for the SSI `exec` directive is not exactly clear. You should be aware that whatever HTML appears in the original HTML request will be thrown away on successful execution of the code being called. This is true for both the `cgi` and `cmd` attributes.

Listing 4-5 shows an example of both attributes.

Listing 4-5. Example Code for the SSI Directives `exec cgi` and `exec cmd`

```
<!DOCTYPE HTML>
<html>
<body id="body">
    <!--#exec cgi="/cgi-bin/example4-3a.py"-->
```

```

</body>
</html>

<!DOCTYPE HTML>
<html>
<body id="body">
    <!--#exec cmd="/cgi-bin/example4-3b.sh"-->
</body>
</html>

```

Note that the only thing the documents in Listing 4-5 do is call the SSI directives `exec cgi` and `exec cmd`. These commands will in turn call a script to output the actual HTML for the request.

Listing 4-6 shows the code for both commands.

Listing 4-6. The Scripts Called by the SSI Directives in Listing 4-5

```

#!/usr/bin/python3

from time import gmtime, strftime

print('Content-type:text/html\r\n\r\n')
print('<!DOCTYPE html>')
print('<html>')
print('<body>')
print('<p>This is output from the example4-3a.py cgi script. ')
print('The current date is ' + strftime("%A %c", gmtime()) +
'.</p>')
print('</body>')
print('</html>')

#!/usr/bin/sh

echo "Content-type:text/html"
echo ""

```

```
echo "<!DOCTYPE html>"
echo "<head>"
echo "    <meta http-equiv=\"Content-Type\" content=\"text/html;
charset=iso-8859-1\" />"
echo "</head>"
echo "<html>"
echo "<body>"
echo "<p>This is output from a shell script. "
echo "The current date is $(date).</p>"
echo "</body>"
echo "</html>"
```

The code in Listing 4-6 is responsible for all the output from each request. The Python script is called for the first request, and the shell script is called from the second request.

Figure 4-3 shows the output from each request.

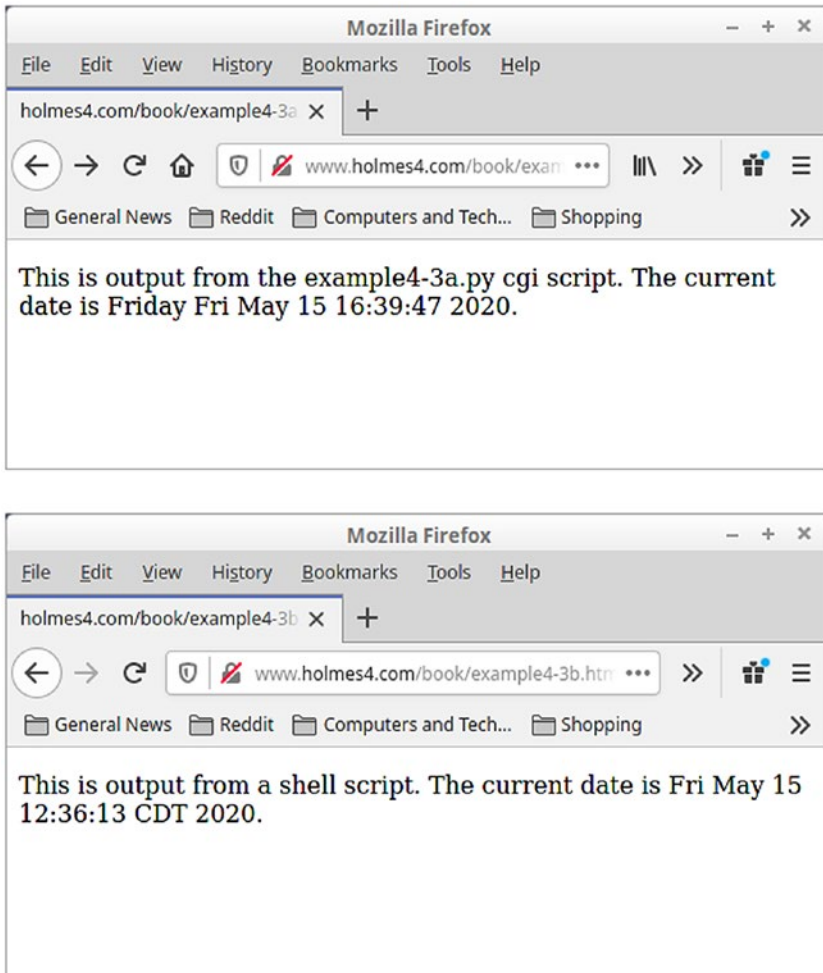


Figure 4-3. Output from the two previous code listings

Figure 4-3 shows the output in the browser. You cannot quite see it, but the request field indicates that this output came from the original HTML request and not the scripts that were called from the HTML. This hides the fact that a CGI or CMD SSI directive was called to actually create the output dynamically. Even if you look at the source for the document, you

cannot tell it did not come from the HTML request. This could be useful in hiding just how the output HTML was created.

You should also note that the `QUERY_STRING` and `CONTEXT_DOCUMENT_ROOT` are the values passed to the original HTML request and remain unmodified when passed to the `cgi` and `cmd` attributes. There could be other environment variables that have original values as well. This can be useful when the directory where the HTML file resides is important to the request.

It is preferable to use the `include` SSI directive instead of the `exec` directive wherever possible as the potential of a security failure goes up when this `exec` directive is used.

The `fsize` SSI Directive

This directive prints the size of the specified file. The output is dependent on the `sizefmt` attribute from the `config` SSI directive. There are two attributes (mutually exclusive) to specify the file, as described here:

file	This is a path/filename relative to the directory containing the current HTML file.
virtual	The %-encoded URL path relative to the <code>DOCUMENT_ROOT</code> from the server config file. If the specification does not begin with a slash (/), it means that it is relative to the current HTML document.

Listing 4-7 contains examples of both `fsize` attributes. They both will appear in the HTML page's output.

Listing 4-7. The HTML Document Containing the fsize Directive

```

<!DOCTYPE HTML>
<html>
<head>
    <title>Example4-4</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body id="body">
    <h1>Example4-4 - fsize SSI Directive</h1>
    <p>The following is an example of the <i>fsize file=</i>
SSI directive. The file size should be presented in either
kilobytes or megabytes.</p>
    <!--#config sizefmt="abbrev"-->
    <!--#fsize file="./example4-4.html"-->
    <p>The following is an example of the <i>fsize virtual=</i>
SSI directive. The file size should be presented in either
kilobytes or megabytes.</p>
    <!--#fsize virtual="./example4-4.html"-->
</body>
</html>

```

Figure 4-4 shows the browser output after requesting this document.

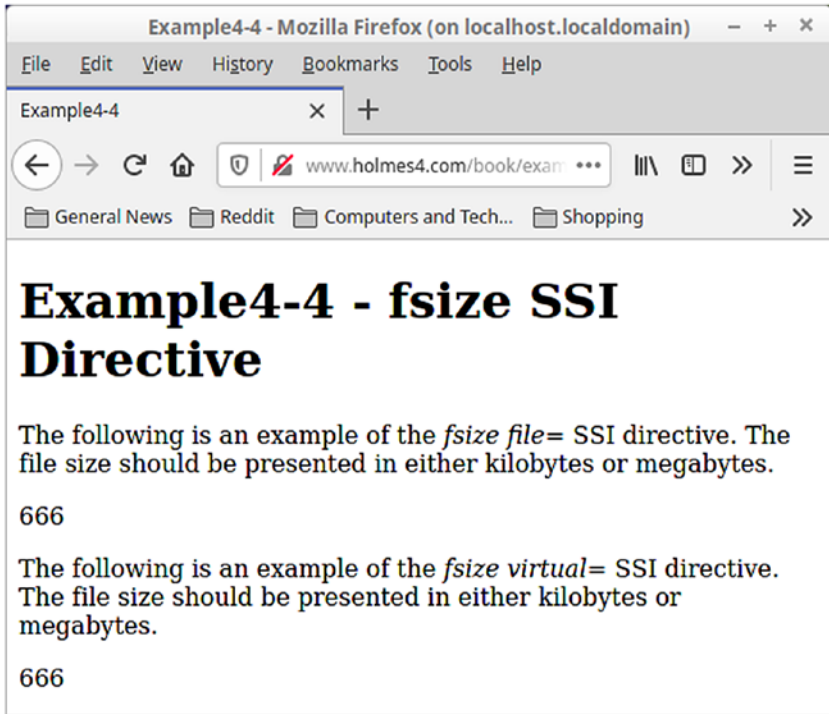


Figure 4-4. Output from Listing 4-7

Output from the *fsize* directive appears inline within the output HTML page. The example in Figure 4-4 has paragraph tags surrounding the size so that it stands out in the example. Also, the size of the original HTML file is small, less than a kilobyte in size, which means there is not a qualifier for the size.

The flastmod SSI Directive

This directive prints the last modification timestamp of the specified file. The output is dependent on the `timefmt` attribute from the `config SSI` directive. There are two attributes (mutually exclusive) to specify the file, as described here:

file	This is a path/filename relative to the directory containing the current HTML file.
virtual	This is the %-encoded URL path relative to the <code>DOCUMENT_ROOT</code> from the server config file. If the specification does not begin with a slash (/), it means that it is relative to the current HTML document.

Listing 4-8 contains examples of both `flastmod` attributes. They both will appear in the HTML page's output.

Listing 4-8. The HTML Document Containing the flastmod Directive

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example4-4</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
</head>
<body id="body">
  <h1>Example4-4 - fsize SSI Directive</h1>
  <p>The following is an example of the <i>fsize file=</i>
    SSI directive. The file
      size should be presented in either kilobytes or
      megabytes.</p>
```

```

<!--#config sizefmt="abbrev"-->
<!--#fsize file="./example4-4.html"-->
<p>The following is an example of the <i>fsize virtual=</i>
SSI directive. The file size should be presented in either
kilobytes or megabytes.</p>
<!--#flastmod virtual="./example4-4.html"-->
</body>
</html>

```

Figure 4-5 shows the browser output after requesting this document.

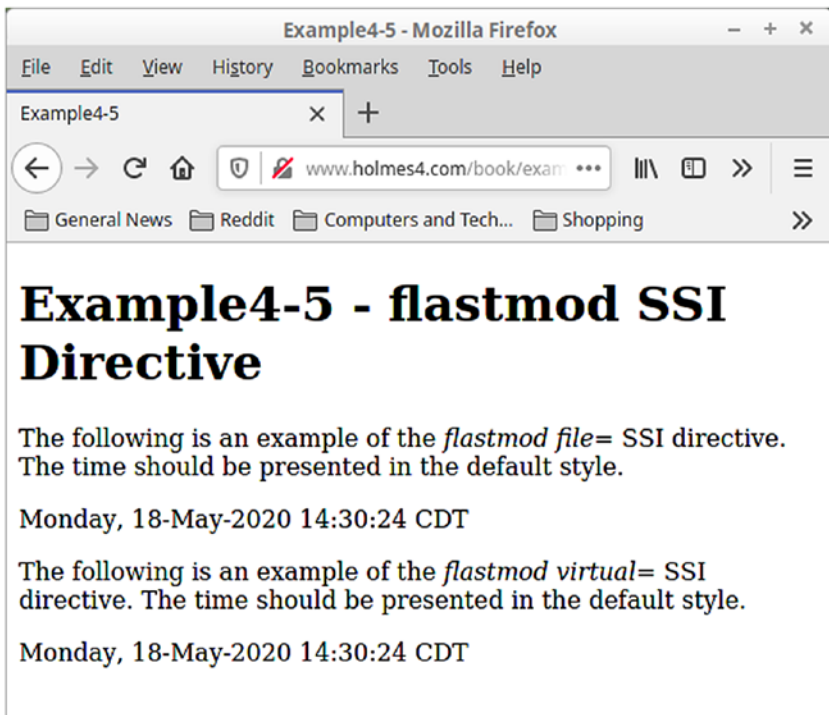


Figure 4-5. Output from Listing 4-8

Output from the `flastmod` directive appears inline within the output HTML page. The example in Figure 4-5 has paragraph tags surrounding the timestamp so that it stands out in the example. The timestamp is shown using the default format.

The include SSI Directive

This directive prints the contents of the file pointed to by the attribute. There are two attributes (mutually exclusive) to specify the file, described here:

file	This is a path/filename relative to the directory containing the current HTML file.
virtual	The %-encoded URL path relative to the <code>DOCUMENT_ROOT</code> from the server config file. If the specification does not begin with a slash (/), it means that it is relative to the current HTML document.

Listing 4-9 contains examples of both `include` attributes. They both will appear in the HTML page's output.

Listing 4-9. The HTML Document Containing the `flastmod` Directive

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Example4-6</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body id="body">
    <h1>Example4-6 - include SSI Directive</h1>
    <p>The following is an example of the <i>include file=</i>
SSI directive.</p>
    <!--#include file="./example4-6a.html"-->
```

```

<p>The following is an example of the <i>include virtual=</i>
i> SSI directive.</p>
<!--#include virtual="/html_lib/example4-6b.html"-->
</body>
</html>

```

Figure 4-6 shows the browser output after requesting this document.

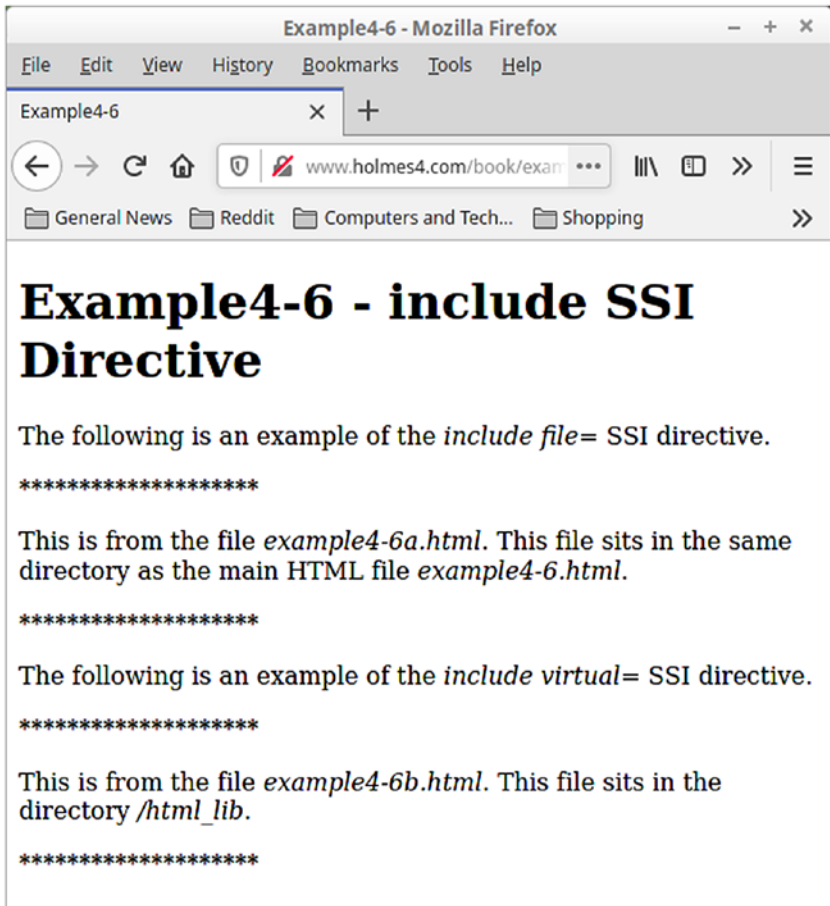


Figure 4-6. Output from Listing 4-8

Output from the `include` directive appears inline within the output HTML page. You should ensure that the included content is a real HTML fragment and not an entire HTML page.

Please note it is not possible to call a script using this directive. Also, the included file may not contain other SSI directives as they will be ignored.

Additional SSI Directives

We will be introducing several SSI directives and concepts in this section because they are all closely related. This will include the `set` directive, additional variables that can be used in your SSI HTML pages, and conditional directives for selecting sets of phrases to be included in your document.

You should note that all the directives and variables described in this section are available in Apache version 2.4 and newer. They may not be available in your HTTP server.

The SSI `set` Directive

The `set` directive is used to set the value of variables. These variables are then available to the conditional directives we will describe later. The following shows an example of this directive:

```
<!--#set var="last_name" value="Ashley"-->
```

The two attributes `var` and `value` describe the name of the variable and its assigned value, respectively. In addition, there are two more attributes that are used only occasionally. They are `encoding` and `decoding` and are used for encoding and decoding the value passed to the variable name. There are several values you can pass in these attributes depending on the encoding you want to translate to/from the `value` attribute. You should see the Apache documentation before attempting to use them.

In addition to setting a literal string to the value, you can also assign some additional variable, which we will describe next. An example of this is shown here:

```
<!--#set var="modified_date" value="$LAST_MODIFIED"-->
```

Note the dollar sign (\$) at the beginning of the variable name. This indicates the variable already exists and the value of the variable should be used. If you need to put a literal dollar sign at the start of your value string, you can escape the dollar sign with the backslash character, as shown here:

```
<!--#set var="amount" value="\$150.00"-->
```

Finally, if you want to put a variable in the middle of a string, you should surround the variable name with curly brackets, as shown here:

```
<!--#set var="doc_name" value=" This document is named  
${DOCUMENT_NAME}."-->
```

One last thing of note is to not use uppercase for your variable names. Apache reserves the right to add new variables to the current list, and those names will be uppercase names.

The current list of available variable names is as follows:

DATE_GMT	This is the server's current timestamp in GMT format.
DATE_LOCAL	This is the server's current timestamp in the server's local format.
DOCUMENT_ARGS	This is the same as the CGI variable QUERY_STRING. If no arguments were passed, this will be the empty string.
DOCUMENT_NAME	This is the filename of the current HTML file.
DOCUMENT_PATH_INFO	This is the same the CGI PATH_INFO variable.

(continued)

DOCUMENT_URO	This is the main HTML DOCUMENT_URI. In the cases of nested includes, this is <i>not</i> the URI of the nested file.
LAST_MODIFIED	This is the last modified timestamp of the main HTML file.
QUERY_STRING_UNESCAPED	This is the unescaped value of the query string.
USER_NAME	This is the username of the owner of the main HTML document.

All of these variables are available to be used in the set directive as well as all the conditional directives.

Listing 4-10 shows examples of all of these variables.

Listing 4-10. An HTML Example Showing How to Use the Available Variables

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example4-7</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
</head>
<body id="body">
  <h1>Example4-7 - Additional Available Variables</h1>
  <p>The following is an example of the <i>DATE_GMT</i> SSI
    variable.</p>
  "<!--#echo var="DATE_GMT"-->"
  <p>The following is an example of the <i>DATE_LOCAL</i> SSI
    variable.</p>
  "<!--#echo var="DATE_LOCAL"-->"
```

```

<p>The following is an example of the <i>DOCUMENT_ARGS</i>
SSI variable.</p>
"<!--#echo var="DOCUMENT_ARGS"-->"
<p>The following is an example of the <i>DOCUMENT_NAME</i>
SSI variable.</p>
"<!--#echo var="DOCUMENT_NAME"-->"
<p>The following is an example of the <i>DOCUMENT_PATH_
INFO</i> SSI variable.</p>
"<!--#echo var="DOCUMENT_PATH_INFO"-->"
<p>The following is an example of the <i>DOCUMENT_URI</i>
SSI variable.</p>
"<!--#echo var="DOCUMENT_URI"-->"
<p>The following is an example of the <i>LAST_MODIFIED</i>
SSI variable.</p>
"<!--#echo var="LAST_MODIFIED"-->"
<p>The following is an example of the <i>QUERY_STRING_
UNESCAPED</i> SSI variable.</p>
"<!--#echo var="QUERY_STRING_UNESCAPED"-->"
<p>The following is an example of the <i>USER_NAME</i> SSI
variable.</p>
"<!--#echo var="USER_NAME"-->"
<br/><p>Note: In the examples above the surrounding
quotation marks are NOT a part of the variable.</p>
</body>
</html>

```

Figure 4-7 shows the output from running this page.

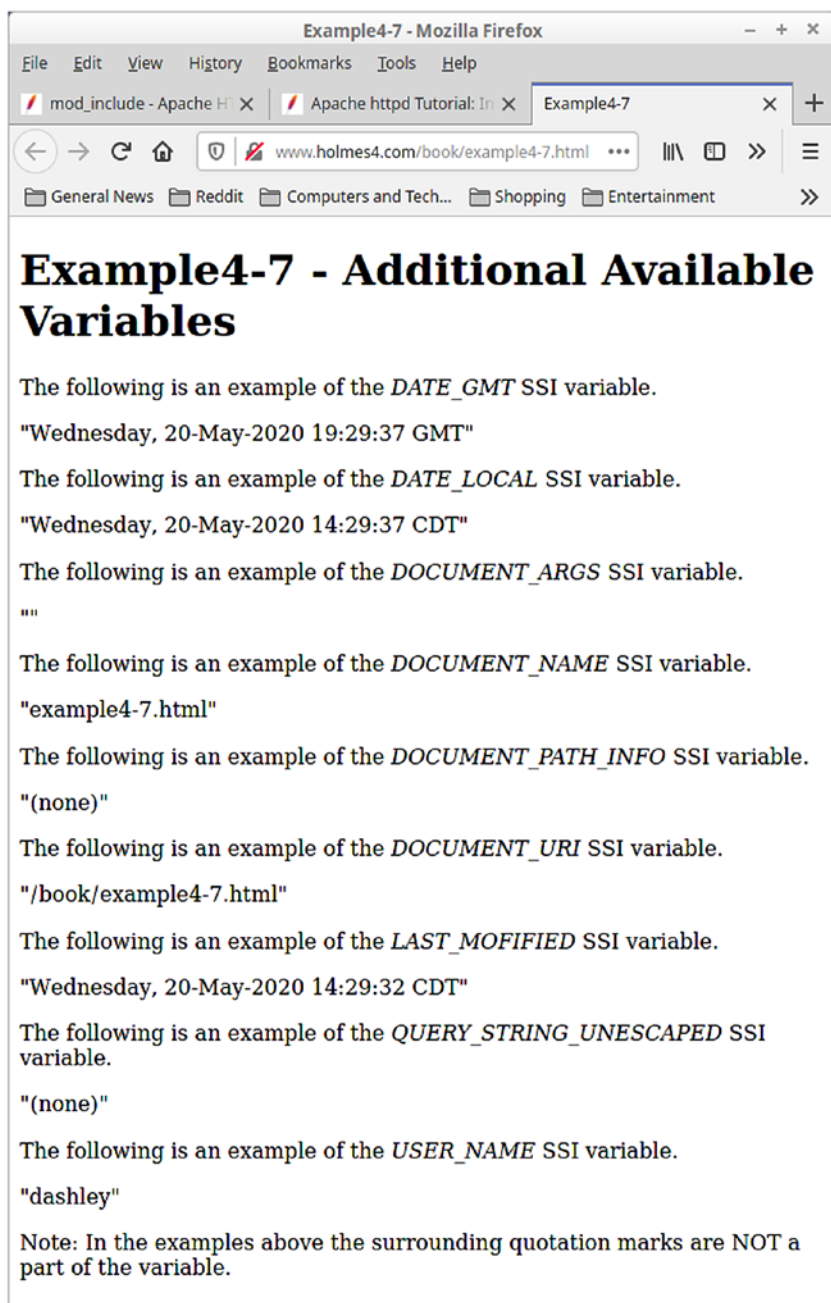


Figure 4-7. The output from Listing 4-10

The SSI Conditional Directives

The SSI conditional directives allow the user to specify HTML text to be included under certain conditions. The directives are `if`, `elif`, `else`, and `endif`. These directives operate in almost the same way as the C/C++ directives with the same names. The `if` directive specifies a condition, and if it is true, the text following it is included in the document. The `elif` directive specifies a different condition that is evaluated if the previous `if` or `elif` condition is false. The `else` condition includes the text that follows it if all previous conditions are false. The `endif` directive ends all conditional text.

 **Warning** Do *not* mix the conditional directives with the previously documented `set` directive. The conditional directives are all evaluated early in the parsing phase of the HTML. The `set` directive (as well as all other SSI directives) is evaluated in a later phase after all the conditional directives have already been evaluated. Thus, the conditional directives have no access to the variables produced by the `set` directive.

Listing 4-11 shows how this works.

Listing 4-11. A Sample HTMP Page Using the Conditional Directives

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example4-8</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
</head>
```

```

<body id="body">
  <h1>Example4-8 - Conditional Directives</h1>
  <p>The following is an example of the SSI conditional
    directives.</p>
  <!--#if expr="%{QUERY_STRING} =~ /^name=([a-zA-Z]+)/"-->
    <!--#if expr="$1 == 'David'"-->
      <p>The Name is correct.</p>
    <!--#else -->
      <p>The Name is NOT correct.</p>
    <!--#endif -->
  <!--#else -->
    <p>The Admin is missing.</p>
  <!--#endif -->
</body>
</html>

```

Figure 4-8 shows one possible output (depending on the calling argument).

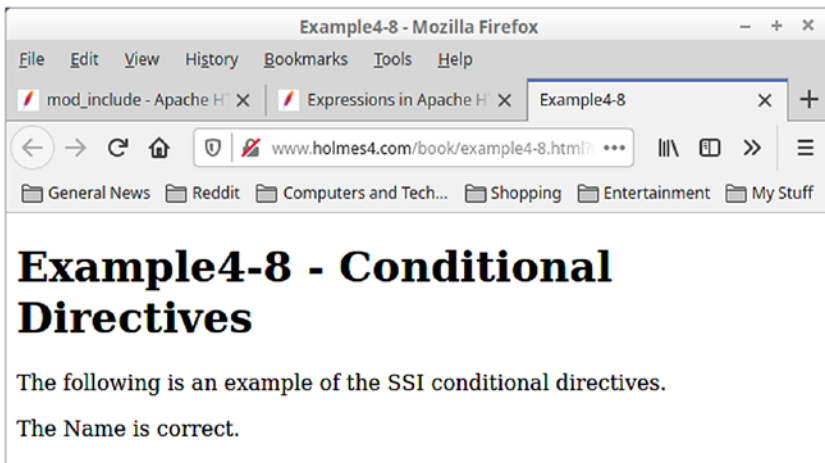


Figure 4-8. One possible output from Listing 4-11

The URL to run the script shown in Figure 4-8 is as follows:

```
http://www.holmes4.com/book/example4-8.html?name=David
```

The previous command is the correct way to obtain a positive result from the script. Any other name or not providing one at all will give a negative result.

The syntax of the conditional directive is simple and matches the other directives we have examined so far. There is only a single attribute, and it is used on the `if` and `elif` conditional directives. The `expr` attribute specifies the conditional expression to be evaluated. The syntax of this condition will be explored a little later. If the condition is true, then the text following the directive will be included. The text can include other conditional directives or any of the SSI directives. Text is included until either an `elif`, `else`, or `endif` SSI directive is found. If the expression is false, all the text is skipped until the next conditional directive is found.

If we examine the condition on the `if` directive, we see that something called the `QUERY_STRING` is being examined for something on the right side of the expression. The `QUERY_STRING` is the same as for the CGI scripts. This is surrounded by `%{...}` to mark it as a known variable. We will list all the known variables for conditional expressions a little later. The right side of the expression is a regular expression. This expression uses the same syntax as regular expressions in Perl 5.0. The middle operator specifies how the variables on each side are to be evaluated. We will examine all the possible operators a little later.

Table 4-1 describes all the available variables for use by the conditional directives. These are the only variables that can be used by the conditional directives.

The tables that follow are all taken from sources on the Web, and any incorrect information is my mistake.

Table 4-1. *SSI Conditional Directive Variables*

Name	Description
HTTP_ACCEPT	Named HTTP request header variable.
HTTP_COOKIE	Named HTTP request header variable.
HTTP_FORWARDED	Named HTTP request header variable.
HTTP_HOST	Named HTTP request header variable.
HTTP_PROXY_CONNECTION	Named HTTP request header variable.
HTTP_REFERER	Named HTTP request header variable.
HTTP_USER_AGENT	Named HTTP request header variable.
REQUEST_METHOD	The HTTP method of the incoming request (e.g., GET).
REQUEST_SCHEME	The scheme part of the request's URI.
REQUEST_URI	The path part of the request's URI.
DOCUMENT_URI	Same as REQUEST_URI.
REQUEST_FILENAME	The full local filesystem path to the file or script matching the request, if this has already been determined by the server at the time REQUEST_FILENAME is referenced. Otherwise, such as when used in virtual host context, the same value as REQUEST_URI.
SCRIPT_FILENAME	Same as REQUEST_FILENAME.
LAST_MODIFIED	The date and time of last modification of the file in the format 20101231235959, if this has already been determined by the server at the time LAST_MODIFIED is referenced.
SCRIPT_USER	The username of the owner of the script.
SCRIPT_GROUP	The group name of the group of the script.

(continued)

Table 4-1. *(continued)*

Name	Description
PATH_INFO	The trailing path name information.
QUERY_STRING	The query string of the current request.
IS_SUBREQ	"true" if the current request is a subrequest, "false" otherwise.
THE_REQUEST	The complete request line (e.g., "GET /index.html HTTP/1.1").
REMOTE_ADDR	The IP address of the remote host.
REMOTE_PORT	The port of the remote host (2.4.26 and later).
REMOTE_HOST	The hostname of the remote host.
REMOTE_USER	The name of the authenticated user, if any (not available during <If>).
REMOTE_IDENT	The username set by mod_ident.
SERVER_NAME	The ServerName of the current vhost.
SERVER_PORT	The server port of the current vhost; see ServerName.
SERVER_ADMIN	The ServerAdmin of the current vhost.
SERVER_PROTOCOL	The protocol used by the request.
DOCUMENT_ROOT	The DocumentRoot of the current vhost.
AUTH_TYPE	The configured AuthType (e.g., "basic").
CONTENT_TYPE	The content type of the response (not available during <If>).
HANDLER	The name of the handler creating the response.
HTTP2	"on" if the request uses http/2, "off" otherwise.
HTTPS	"on" if the request uses https, "off" otherwise.

(continued)

Table 4-1. *(continued)*

Name	Description
IPV6	"on" if the connection uses IPv6, "off" otherwise.
REQUEST_STATUS	The HTTP error status of the request (not available during <If>).
REQUEST_LOG_ID	The error log ID of the request (see <code>ErrorLogFormat</code>).
CONN_LOG_ID	The error log ID of the connection (see <code>ErrorLogFormat</code>).
CONN_REMOTE_ADDR	The peer IP address of the connection (see the <code>mod_remoteip</code> module).
CONTEXT_PREFIX	Tells you how the server used an <code>Alias</code> directive (or a similar feature, like <code>mod_userdir</code>) to translate the URL path to the file system path.
CONTEXT_DOCUMENT_ROOT	Tells you how the server used an <code>Alias</code> directive (or a similar feature, like <code>mod_userdir</code>) to translate the URL path to the file system path.
TIME_YEAR	The current year (e.g., 2010).
TIME_MON	The current month (01, ..., 12).
TIME_DAY	The current day of the month (01, ...).
TIME_HOUR	The hour part of the current time (00, ..., 23).
TIME_MIN	The minute part of the current time.
TIME_SEC	The second part of the current time.
TIME_WDAY	The day of the week (starting with 0 for Sunday).
TIME	The date and time in the format 20101231235959.
SERVER_SOFTWARE	The server version string.
API_VERSION	The date of the API version (module magic number).

The variables listed in Table 4-1 must always be specified in uppercase. The SSI processor is case sensitive, so keep that in mind as you write your HTML.

In addition to the variables listed in Table 4-1, you may also include quoted text as a constant. Constants can be surrounded with either single (') or double (") quotes.

Table 4-2 lists all the binary operators that may be used in conditional directives.

Table 4-2. *Conditional Directive Binary Operators*

Name	Alternative	Description
==	=	String equality
!=		String inequality
<		String less than
<=		String less than or equal
>		String greater than
>=		String greater than or equal
=~		String matches the regular expression
!~		String does not match the regular expression
-eq	eq	Integer equality
-ne	ne	Integer inequality
-lt	lt	Integer less than
-le	le	Integer less than or equal
-gt	gt	Integer greater than

(continued)

Table 4-2. *(continued)*

Name	Alternative	Description
-ge	ge	Integer greater than or equal
-ipmatch		IP address matches address/netmask
-strmatch		Left string matches pattern given by right string (containing wildcards *, ?, [])
-strcmatch		Same as -strmatch, but case insensitive
-fnmatch		Same as -strmatch, but slashes are not matched by wildcards

Table 4-3 lists all the unary operators used in conditional directives.

Table 4-3. *Conditional Directive Unary Operators*

Name	Description
-d	The argument is treated as a filename. True if file exists and is a directory.
-e	The argument is treated as a filename. True if the file (or dir or special) exists.
-f	The argument is treated as a filename. True if the file exists and is a regular file.
-s	The argument is treated as a filename. True if the file exists and is a regular file.
-L	The argument is treated as a filename. True if the file exists and is a symlink.
-h	The argument is treated as a filename. True if the exists and is a symlink (same as -L).

(continued)

Table 4-3. *(continued)*

Name	Description
-F	True if string is a valid file, accessible via all the server's currently configured access controls for that path. This uses an internal subrequest to do the check, so use it with care; it can impact your server's performance.
-U	True if string is a valid URL, accessible via all the server's currently configured access controls for that path. This uses an internal subrequest to do the check, so use it with care; it can impact your server's performance.
-A	Alias for -U.
-n	True if string is not empty.
-z	True if string is empty.
-T	False if string is empty, "0", "off", "false", or "no" (case insensitive). True otherwise.
-R	Same as "%{REMOTE_ADDR} -ipmatch . . .", but more efficient.

At this point, it should be noted that there are no OR or AND operators. This means that only simple comparisons can be made in an `if` directive. Multiple comparisons must be made by nesting `if` directives.

Table 4-4 lists all the functions available for use in comparisons.

Table 4-4. *Conditional Directive Functions*

Name	Description	Notes
req, http	Gets HTTP request header; header names may be added to the Vary header	
req_novary	Same as req, but header names will not be added to the Vary header	
resp	Gets HTTP response header (most response headers will not yet be set during <If>)	
reqenv	Looks up request environment variable (as a shortcut, v can also be used to access variables)	Ordering
osenv	Looks up operating system environment variable	
note	Looks up request note	Ordering
env	Returns first match of note, reqenv, osenv	Ordering
tolower	Converts string to lowercase	
toupper	Converts string to uppercase	
escape	Escapes special characters in %hex encoding	
unescape	Unescapes %hex-encoded string, leaving encoded slashes alone; return empty string if %00 is found	
base64	Encodes the string using base64 encoding	
unbase64	Decodes base64-encoded string; returns truncated string if 0x00 is found	
md5	Hashes the string using MD5, then encodes the hash with hexadecimal encoding	
sha1	Hashes the string using SHA1, then encodes the hash with hexadecimal encoding	

(continued)

Table 4-4. *(continued)*

Name	Description	Notes
file	Reads contents from a file (including line endings, when present)	Restricted
filemod	Returns last modification time of a file (or 0 if file does not exist or is not a regular file)	Restricted
filesize	Returns size of a file (or 0 if file does not exist or is not a regular file)	Restricted

The functions marked “Restricted” in the final column are not available in some modules like `mod_include`.

The functions marked “Ordering” in the final column require some consideration for the ordering of different components of the server. You should remember that the `if` directive is evaluated early in the response processing.

The following are some sample `if` directives to give you an idea of what is possible:

```
<!-- Compare the host name to example.com -->
<!--#if %{HTTP_HOST} == 'example.com' -->

<!-- Force text/plain if requesting a file with the query
string contains 'forcetext' -->
<!--#if %{QUERY_STRING} =~ /forcetext/ -->

<!-- Check a HTTP header for a list of values -->
<!--#if %{HTTP:X-example-header} in { 'foo', 'bar', 'baz' } -->

<!-- Check an environment variable for a regular expression,
negated. -->
<!--#if ! reqenv('REDIRECT_FOO') =~ /bar/ -->
```

```
<!-- Check against the client IP -->  
<!--#if -R '192.168.1.0/24' -->  
  
<!-- Function example in boolean context -->  
<!--#if md5('foo') == 'acbd18db4cc2f85cedef654fccc4a4d8' -->
```

Summary

Using SSI can be as easy or complicated as you want to make it. While it does not fit every situation, it is a useful tool for easing your web server maintenance. Used under the right circumstances, it can make everything easier.

This chapter has introduced all the concepts needed to make SSI work for you whatever your needs. Just remember that the main SSI directives are evaluated after the SSI conditional directives.

CHAPTER 5

Using Flask and Jinja

Flask and Jinja are the main two components of a Python system to manage a dynamic website. This system has similarities to other systems like Django. All of these systems consist of a web server and two Python components—a management component and a dynamic web page component. The idea behind this system is that the website should not be constrained to only one way of solving a problem. The designer should be able to choose the components that work for their situation.

Depending on the needs, other components could be added into the mix. These might be components such as a database, a document management system, a video playback system, or any other component to manage something connected to a computer. Of course, most website designers would prefer that any system connected to the website had a Python interface. But if a Python interface is not available, other drivers could be used, provided that Python interfaces could be developed.

All of this means a website designer has choices. Countless successful websites have been designed that utilize these components. Obviously, website designers have embraced the choices that have been provided by utilizing these components. But there are drawbacks. This chapter will examine the advantages and the disadvantages of using Flask and Jinja as well as suggest some solutions for the weaknesses you will encounter.

This chapter is divided into two major sections. In the first section, you will learn about the Web Server Gateway Interface (WSGI) and Flask components and how to install and use them. In the second section, I will demonstrate the Jinja2 template component and how to create dynamic web pages.

WSGI and Flask

WSGI and Flask are two components that work together to provide a powerful mechanism to create and manage dynamic web pages. In this section, I will discuss how to install and test each component. Our test system in this chapter uses Fedora Linux and the Apache HTTP Server, which are pretty common in the real world. But if your system runs a different operating system or web server, the instructions will be different. I will try to provide options for you in that case, but they may not be complete.

The Flask component requires that WSGI be installed to work at all. So, our first examples will involve installing and testing WSGI.

Installing and Testing WSGI

WSGI can be installed in two ways. Your operating system may provide an installable package for WSGI, or you may have to install the latest WSGI source code and compile it yourself. In both cases, you will still need to configure it using the web server configuration file, which we will discuss a little later in the chapter.

If your operating system provides a WSGI package, I highly recommend that you install and use that package since it is already compiled specifically for your operating system. For Windows users, your only real option for a web server is Apache. The source for WSGI assumes that Apache and its development environment are already installed on your system. You have the choice of downloading the source or installing a prebuilt version via `pip`.

To install using `pip`, run the following command:

```
pip install mod_wsgi
```

This will install the latest release of `mod_wsgi`, so all you have left is to configure it in the Apache configuration file. There are more complete instructions at <https://pypi.org/project/wsgi>.

If your system has an installable `mod_wsgi` package, you should use it if it meets your requirements. Our test system is a 64-bit Fedora system running Apache 2.4 and Python 3.6. You should make sure that whatever versions of Python, web server, and `mod_wsgi` you are trying to use are compatible with each other. That was easy with Fedora since an installable package was available. To install `mod_wsgi` on Fedora, use the following command:

```
sudo dnf install python3-mod_wsgi
```

This will install the `mod_wsgi` module so it can be used by Apache. After this, we still need to configure it for use by the Apache web server. After the install, you should make sure that some configuration has been done. First, make sure the following file has been created (Linux only):

```
/etc/httpd/conf.modules.d
```

This file will ensure that the WSGI module has been installed into Apache. Next, you will need to perform some manual configuration either on the main server thread config file or on a virtual server file located in `/etc/httpd/conf.d`. The configuration for WSGI looks something like this:

```
WSGIScriptAlias /wsgitest /pub/www/holmes4/docroot/wsgitest/
wsgitest.wsgi
<Directory "/pub/www/holmes4/docroot/wsgitest">
    Require all granted
</Directory>
```

The first line specifies the Python script to be run. There is a lot about this line that we need to understand as it impacts a lot about the script to be run. The first parameter, `/wgitest`, has two functions and supplies the partial URL that will invoke the script. The second parameter specifies the location and name of the WSGI Python script to be run. There are two items to note about this parameter. First, the location specified does *not* have to be in the main web server file system. It can be anywhere on your file system. The second point is that the Python script can be named anything you want as long as it has a `.wsgi` extension. The other point is that the name of the file does not need to match the first parameter. Again, it can be named anything you want. Our example used the same name to keep things simple, but you do not have to follow this convention.

The next three lines configure the server permissions for the directory where the Python script resides. The permission shown will need to be specified whether or not your location is inside the web server file system.

The last item we need to discuss is not obvious with this example. You can have as many `WSGIScriptAlias` lines as you need. These may all use the same location or various locations. Just remember that each location will need a `Directory` specification to give them the proper permissions.

Once you have a complete configuration for WSGI, you should reboot your server to make it active. Now we need to create a test Python script to test our configuration. In our case, we need to create the `wgitest` directory off the root of the web server's main file system. Now we need to create our `wsgitest.wsgi` script file. Our `wsgitest.wsgi` script looks something like Listing 5-1.

Listing 5-1. `wsgitest.wsgi`

```
import sys
import os

def application(environ, start_response):
    status = '200 OK'
```

```

output = ''
output += "sys.version = %s\n" % repr(sys.version)
output += "sys.prefix = %s\n" % repr(sys.prefix)
output += "path = %s\n" % os.environ['PATH']
output += "REQUEST_METHOD = %s\n" % environ['REQUEST_METHOD']
name = 'SCRIPT_NAME'
if name in environ:
    output += name + " = %s\n" % environ[name]
else:
    output += name + " = \n"
name = 'PATH_INFO'
if name in environ:
    output += name + " = %s\n" % environ[name]
else:
    output += name + " = \n"
name = 'QUERY_STRING'
if name in environ:
    output += name + " = %s\n" % environ[name]
else:
    output += name + " = \n"
name = 'CONTENT_TYPE'
if name in environ:
    output += name + " = %s\n" % environ[name]
else:
    output += name + " = \n"
name = 'CONTENT_LENGTH'
if name in environ:
    output += name + " = %s\n" % environ[name]
else:
    output += name + " = \n"
output += "SERVER_NAME = %s\n" % environ["SERVER_NAME"]

```

```

output += "SERVER_PORT = %s\n" % environ["SERVER_PORT"]
output += "SERVER_PROTOCOL = %s\n" % environ["SERVER_PROTOCOL"]
(v1, v2) = environ["wsgi.version"]
output += "wsgi.version = %s\n" % str(v1) + '.' + str(v2)
output += "wsgi.url_scheme = %s\n" % environ["wsgi.url_scheme"]
output += "wsgi.multithread = %s\n" % environ["wsgi.
multithread"]
output += "wsgi.multiprocess = %s\n" % environ["wsgi.
multiprocess"]
output += "wsgi.run_once = %s\n" % environ["wsgi.run_once"]

response_headers = [('Content-type', 'text/plain'),
                    ('Content-Length', str(len(output)))]
start_response(status, response_headers)

return [output.encode('utf-8')]

```

There are a few items we need to discuss about this listing. First, this is a Python file, but there is no `#!/usr/bin/python` specification at the top of the file. It is not needed since the only kind of file a WSGI can invoke is a Python script. The next thing to note is that the single function contained in the file is named `application`. This is the one and only function name that will be invoked by WSGI. The script has no main line statements, only the single function.

The main body of the function creates the body of the HTML page. In our case, we are just returning some text, so we don't enclose that in any kind of HTML formatting.

Next, we need to look at the last three statements of the function. The first statements build the response header for the web page. In a lot of cases, this is placed at the top of the response to the requestor. WSGI provides functionality to pass the necessary response headers back to the caller. The next statement calls that functionality. The main thing to note is that we are supplying the length of the response. Normally we do not

fill in this entry because we are responding with an HTML file. Since ours is a text file, we need to explicitly supply this entry. Otherwise, the web browser parser will not be able to determine where the file ends. If it were HTML, the parser would stop parsing when it finds the `</html>` tag.

The last line returns the response we built earlier in the script. Note that we have to encode our response. This is a weird requirement you may have never seen before. But WSGI will accept only encoded responses, not plain text. Also, note that the returned response is enclosed in square brackets so that the encoded string is returned as a series of bytes.

★ **Important** No one in their right mind creates native WSGI web pages. WSGI is a foundation for other systems such as Flask, Django, and many others. The previous example is only a simple test to ensure that WSGI is working as designed.

You can run this script with the following URL from your web browser:

<http://www.holmes4.com/wsgitest>

Just substitute your web server name and WSGI application name in the previous command.

Figure 5-1 shows the output from the Python script in Listing 5-1.

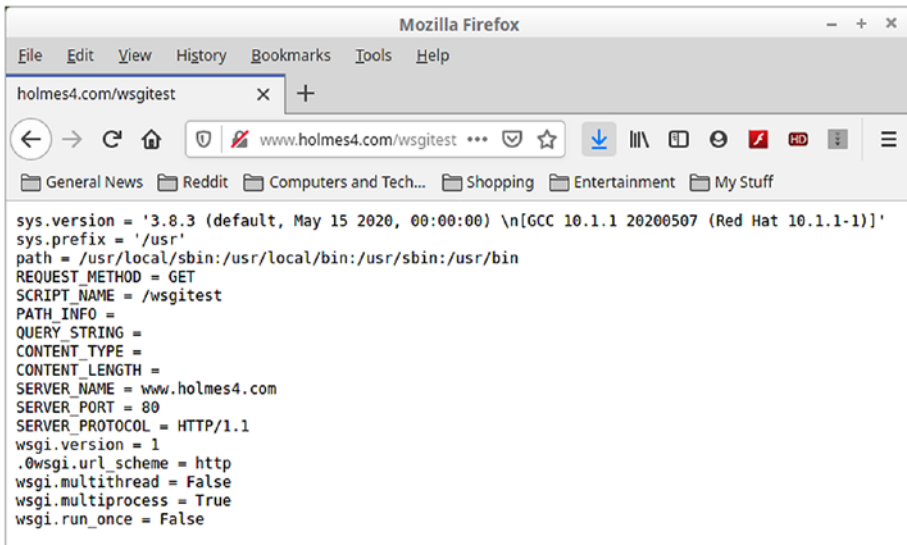


Figure 5-1. Output from the *wsgitest.wsgi* Python application

The web page in Figure 5-1 gives you some idea of the available environment entries to the WSGI script. As you can see, it is pretty limited but does have some useful information. The Flask environment has a lot of additional values available, which is why you want to use it as your application environment instead of native WSGI.

Installing and Testing Flask

Flask is a Python module that is not usually available in your operating system's repository. You must install it yourself. This turns out to be easy as only one command is needed to get it installed, shown here:

```
sudo pip install flask
```

Beware, if you do not install it as root or do not use the `sudo` prefix command, the library will be installed in `/usr/local/lib64/python3.8/site-packages` on Linux. Unfortunately, this location is *not* available to

programs running inside the web server, so Flask is not available. If this is a problem for you, there are two choices available. You can copy or move the libraries from their current location to `/usr/lib64/python3.8/site-packages` or you can create links to the packages in that location. One thing to note is that the Python version in the previous paths may not match your version of Python. If so, then just change the path accordingly. Windows users should not run into this problem.

Once the installed libraries are in the correct directory, the install is complete. To get Flask working, you need to perform a few more steps in Apache.

Just as we needed to configure WGI for use by Apache, we also need to perform a similar step for Flask. The first step is to add one new line to the Apache configuration file.

```
WSGIScriptAlias /flasktest /pub/www/holmes4/docroot/wsgitest/
flasktest.wsgi
```

This should be familiar to you as it has a similar format to the WSGI entry in the Apache config file. Just add this line before or after the similar WSGI line.

Next, we need to create the `flasktest.wsgi` file specified in the previous line at the location specified. Listing 5-2 shows what should go into that file.

Listing 5-2. The `flasktest.wsgi` File Contents

```
import logging
import sys

logging.basicConfig(stream=sys.stderr)
sys.path.insert(0, '/pub/www/holmes4/docroot/wsgitest/')
from flasktest import app as application
application.secret_key = 'flask test on apache 2.4'
```

As you can see, this is not the main program. This is some setup information for your real Flask application. The `sys.path.insert()` function is where our real Python program is located. The next statement imports our application file (still to be created) and the function it uses. The last line registers the secret key for the application. This string can be anything you want as long as it is of nonzero length. We use a short description of the Python application to be executed. An examination of this file should reveal that your program can reside anywhere. All you have to do is alter the location in the `sys.path.insert()` statement.

One thing to note about this file is that it does not invoke the registered function. Flask does that for you for the same reason that this is broken up between two files. It protects your application from poisoning the Flask environment since it is called from Flask. Your program will have its own variable space that is not shared with Flask.

The last task we need to perform is to create our actual program. Listing 5-3 shows a simple test program that we will use to ensure that Flask is operating correctly.

Listing 5-3. The flasktest.py File Contents

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello world!"

@app.route("/second")
def hello2():
    return "Hello second!"
```

Wow, is this all there is? Yes. In fact, it is more than first meets the eye. There are actually two executable functions included that can be invoked from the browser command line. The first function can be invoked with the following browser command:

<http://www.holmes4.com/flasktest>

This should cause the returned "Hello world!" string to appear in the browser. The second function can be invoked as follows:

<http://www.holmes4.com/flasktest/second>

This should cause the returned "Hello second!" string to appear in the browser. If both of these invocations work, then you have successfully created your first Flask application.

The URL for invoking the application is made up of three parts. The first is the name of the web server. The next part is the name of the application to invoke. The last part is optional and specifies a different function to be invoked, the `hello2` function in our case. All this is specified by the `@app.route()` decorator that precedes each function. This allows you to have multiple functions contained within your application, all available from the `flasktest` base name. The decorator supersedes the application name from the `flasktest.wsgi` file.

There is a lot of power provided with Flask that we have not even touched on. Before we get to that, I will show you how to build a nice HTML page using Flask. To save time, I will show that some CGI programs can be converted to Flask with very little work. To do that, we will convert Listing 3-8 to run under Flask.

The only changes to Listing 3-8 we need to make are to place the entire program under a function in the `flasktest.py` file. That makes the top of the file look like the following:

```
from flask import Flask
import os, time, string

app = Flask(__name__)
```

```

@app.route("/")
def hello():
    return "Hello world!"

@app.route("/second")
def hello2():
    return "Hello second!"

@app.route("/pictures")
def hello3():

```

As you can see, we just added a new function to contain the new application. Next, we need to alter the code that obtains the `DOCUMENT_ROOT`. This CGI value is not available in a Flask environment. If you need it like this program does, then you will have to infer it from the location of the Python script. Since the location of our Flask program is inside the web file server file system, we can obtain the `DOCUMENT_ROOT` by stripping off the program filename and the directory that contains it from the `__file__` variable to give us the root location of the web server file system. The following code fragment shows how we accomplished that:

```

# get the document base for the html_lib directory
(docroot, tail) = os.path.split(__file__) # split off the
file name
(docroot, tail) = os.path.split(docroot)  # split off the
first directory
html_lib_path = docroot + '/html_lib/'

```

The last change we need to make is to the last statement in the Python script. Instead of writing the HTML directly to the browser client, we return it to Flask, and in turn Flask will write it for us.

```

# return the substituted lines
    return html

```

That is all we needed to change. We now should have a working Flask program that has the same functionality and display graphics of the original CGI program.

 **Warning** To execute the program from the browser, you will first have to restart your web server. Otherwise, Flask will not know there is a new function in the `flasktest.py` file. The reason for this is that Flask examines each file registered via the `WSGIScriptAlias` in the web server configuration file at web server startup. If you add a new Python function to a file controlled by WSGI, it will be unknown until you restart the web server.

Jinja2

Jinja2 is a Python module that performs text substitution in almost any text file. It can work on plain ASCII, HTML, CSV, and practically any kind of text file. It is just one member of a family of text substitution frameworks available for Python. All of these frameworks have good and bad points, but it cannot be denied that they can improve your application development. Jinja2 is the logical choice for Flask because it is installed with Flask. Since you have it already, you might as well make use of it.

The idea behind a text substitution framework is to separate the display parts of your program from the code logic. This is a modern approach to software development. Although that is a great approach when developing software, sometimes practical approaches are needed to actually make something work. This is true for all text substitution programs that try to do more than just simple text substitution.

The Python `string` module does simple text substitution, swapping one value for another. We used this in the CGI examples discussed previously. But in more complicated situations where we have repeating

information, we need a more powerful substitution mechanism. This is especially true with database information where we need to display row after row of similar data. This would be hard to accomplish with a simple text substitution mechanism. But with a more powerful mechanism, we can actually program a loop in the source file to substitute multiple rows from a database table. This is a huge productivity tool for that type of data.

But to start with, let's look at a simple text substitution mechanism example. Listing 5-4 shows the combined power of Flask and Jinja2.

Listing 5-4. A Simple Flask and Jinja2 Example

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
@app.route('/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

This simple program has a lot going on, most of which is not really obvious. First, we are importing an additional module named `render_template`. This is a Jinja2 module that takes a template name and some additional parameters specifying the variables and values to be used for substitution in the template. This is shown in the last line of the program. But you may be asking, “Where is the template?” By default, the Jinja2 template is located in a subdirectory named `templates` just under the directory holding the program. We will show this template in Listing 5-5.

The next parts are the two decorator statements for the function. The first one is one we have seen before and requires no explanation. The second decorator has an additional strange entry inside greater-than and less-than symbols. This specifies that the name is optional. That means it may or may not appear in the invocation command. Thus, the browser command line may have one of the following two forms:

<http://www.holmes4.com/jinjatest/>
<http://www.holmes4.com/jinjatest/David>

The second example has a name attached to the end. The template contains logic to determine if the name was given or not and respond accordingly.

The function is declared so that the name has a default value. Ours uses None as the default value, but we could have specified a default name.

The last statement invokes the template to be rendered and adds extra values needed by the template to substitute the necessary variables in the template.

So, let's examine the template so we can see how this all fits together.

Listing 5-5. The Simple Flask and Jinja2 Example Template

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello, World!</h1>
{% endif %}
```

This is just enough HTML to show how all this functionality works. The statements we are interested in are the ones enclosed with `{% ... %}`. These are not HTML; instead, they are instructions to Jinja2. The first statement tests to see whether the variable name has a value (besides None). If it does, then the following HTML statement is included in the output. The next Jinja2 statement is the else part of the test. It includes the subsequent HTML statement if the else condition is true. The last Jinja2 statement ends the test block.

Under the true portion of the Jinja2 block, we see a statement enclosed in `{{ expression }}`. This contains an expression that will be evaluated and then included in the output.

Figure 5-2 shows the output sent to the browser when a name is included on the browser command line.

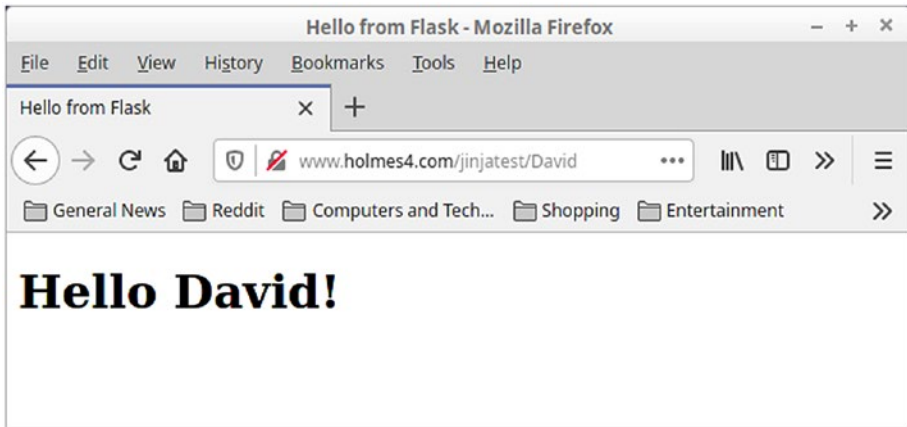


Figure 5-2. *Output of the simple Flask and Jenja2 example*

As you can see, including a name on the command line changes the output from “Hello World!” to “Hello David!” This is a powerful mechanism that allows you to use a single HTML page for all users.

There are four major types of Jinja2 statements available to you. These are listed here:

<code>{% ... %}</code>	For statements
<code>{{ ... }}</code>	For expressions
<code>{# ... #}</code>	For comments
<code># ... ##</code>	For line statement

Statements are similar to Python statements. We will see an example of this kind of statement in the next example.

Expressions are similar to Python expressions.

Comment statements are just that, comments. They will not be included in the HTML output.

Line statements are for single-line statements. It is highly recommended that you do not utilize this type of statement because it is usually confusing to someone not familiar with Jinja2.

There are many expressions and loops that can be created with Jinja2. They are all documented on the website at <https://jinja.palletsproject.com/en/2.11.x/>.

Our next example is a little more extensive (Listing 5-6). It shows how to display the results of a database query when the total number of output lines is unknown. To keep things simple, the actual database query is not included, only the output.

Listing 5-6. Database Query Display Example

```
from flask import Flask, render_template
from jinja2 import Template, Environment, PackageLoader,
select_autoescape
import os, time

app = Flask(__name__)

def get_rows():
    rows = list()
    rows = ["DEPTNO DEPTNAME                MGRNO  ADMRDEPT
LOCATION          ",
           "-----",
           "A00   SPIFFY COMPUTER SERVICE DIV.    000010 A00   -",
           "B01   PLANNING                          000020 A00   -",
           "C01   INFORMATION CENTER                000030 A00   -",
           "D01   DEVELOPMENT CENTER                -      A00   -",
           "D11   MANUFACTURING SYSTEMS            000060 D01   -",
           "D21   ADMINISTRATION SYSTEMS           000070 D01   -",
           "E01   SUPPORT SERVICES                 000050 A00   -",
           "E11   OPERATIONS                      000090 E01   -",
```

```

        "E21    SOFTWARE SUPPORT          000100 E01    -",
        "F22    BRANCH OFFICE F2         -      E01    -",
        "G22    BRANCH OFFICE G2         -      E01    -",
        "H22    BRANCH OFFICE H2         -      E01    -",
        "I22    BRANCH OFFICE I2         -      E01    -",
        "J22    BRANCH OFFICE J2         -      E01    -",
        "",
        "  14 record(s) selected."
    ]
    return rows

@app.route('/')
def department(name=None):
    # create the substitutable content
    pagedict = dict()
    title = 'Table Display'
    title1 = 'Department Table'
    pagedict['title'] = 'Table Display'
    titleleft = 'Department Table'
    asctime = time.asctime()
    parts = asctime.split()
    year = parts[4]
    rows = get_rows()

    # get the template and make the substitutions
    html = render_template('jinatest2.html', rows=rows,
                           title=title, \
                               titleleft=titleleft, title1=title1,
                               year=year)

    # return the substituted lines
    return html

```

The `get_rows()` function returns the database rows to be displayed. Normally this would include a query and row fetching from the database, but we did not want to clutter up the listing with stuff we are not really covering with this book.

The `department()` function is our main function. It is simple. It gathers some substitution variables and the row data. It then calls the `render_template()` Jinja2 function to render the template. Finally, it returns the rendered HTML to Flask.

The template file is also stored in the templates directory, which is just below the directory containing the Python program.

The template itself is similar to the HTML we have seen before except for the substitution variables (see Listing 5-7).

Listing 5-7. HTML Template for Listing 5-6

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ title }}</title>
  <meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
  <link rel="stylesheet" type="text/css" href="/css/
  example2-2.css" />
</head>
<body>
<!--Begin Header-->
<div id="header">
  <!-- Begin Picture Column -->
  <div id="header-left">
    <a href="/"></a>
    <br />image by W.David Ashley
```

```

    </div>
    <!-- Begin Title Column -->
    <div id="header-right"><h1>{{ title1 }}</h1>
    </div>
</div>
<!--Begin Page Menu-->
<div id="pagemenu">
    <!-- Begin Left Title Column -->
    <div id="pagemenu-left">{{ titleleft }}
    </div>
    <!-- Begin Page Menu Column -->
    <div id="pagemenu-right"><a href="/">&nbsp;Home&nbsp;</a>&nbsp;</
a>&nbsp;<br>
        <a href="/">&nbsp;Menu2&nbsp;</a>&nbsp;<br>
        <a href="/">&nbsp;Menu3&nbsp;</a>&nbsp;<br>
    </div>
</div>
<!--Begin Content-->
<div id="content">
    <!-- Begin Content Menu Column -->
    <div id="content-left">
    </div>
    <!-- Begin Content Column -->
    <div id="content-right">
        <pre>{% for row in rows %}{{ row|e }}<br/>{%
        endfor %}</pre>
    </div>
<!-- Begin Footer -->
<div id="footer">
    <div id="footer-data">

```

```

    
    <br />&#169; Copyright 2010-{{ year }}
    W. David Ashley. All rights reserved.
</div>
</div>
</body>

```

The substitutable variables are the same syntax as used in the previous example. However, the query row display is new and needs explanation. The expression `{% for row in rows %}` is a looping expression. It loops through each row of data, and the syntax should be familiar to you as it derives from Python. The next expression, `{{ row|e }}`, displays a single row of data for each loop iteration. The next expression, `{% endfor %}`, ends the loop.

To invoke this example, use the following browser command:

<http://www.holmes4.com/jinjatest2>

This command will produce a formatted HTML page with the output in a nice display.

To confirm the loop works as expected, the output to the browser is displayed in Figure 5-3.

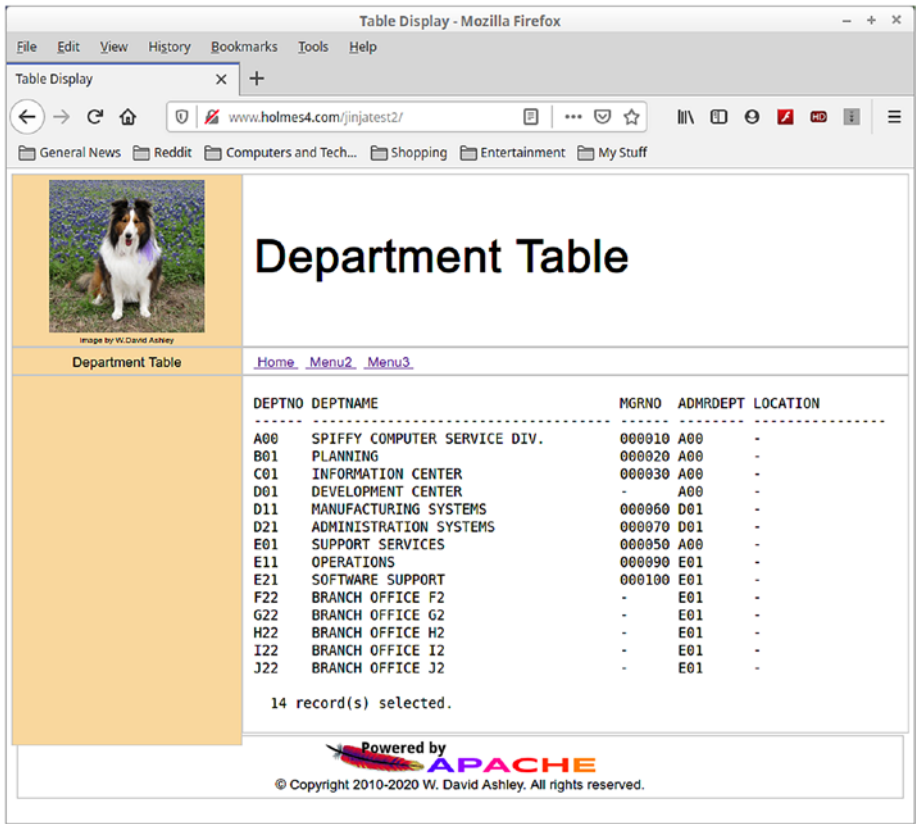


Figure 5-3. Output from Listing 5-6 and 5-7

The output is just as we expected it. This shows the real power of Jinja2 where we do not have to make changes to the program or the HTML to accommodate variable amounts of data in our HTML page.

There are one or two “gotchas” that you can run into using Jinja2. First, we had to hard-code the HTML menus into the HTML page. The reason for this is that the default in Jinja2 is to escape any text substituted into the document. If we tried to insert HTML tags into our template, then they would be escaped (the GT and LT symbols would be replaced). Thus, our menus would not only look strange, they would not work at all. This behavior can be changed, but only globally.

The other point has to do with the display of the lines. You should note that the expressions for the loop all appear on a single line with a single `<br/>` tag inside the loop. If you do not format it this way, you will invariably get blank lines between each line of real output. All the documentation for this loop structure shows each instruction on separate lines, and this always produces unwanted extra blank lines in the output.

Summary

The combination of Flask and Jinja2 makes a wonderful productivity tool for both small and large projects. The extensive built-in functions and formatting options available in Jinja2 make for solutions for almost every problem you may encounter. The number of available expressions and functions available with Jinja2 along with the available extensions make a powerful development environment that can create a useful web interface.

However, you may find that the expressions available violate some aspects of object-oriented programming (OOP), especially the principle of keeping the display of data separate from the program logic. Having expressions within the HTML is a clear violation of this principle.

But, if you can live with these drawbacks, Flask and Jinja2 may be an acceptable solution for your situation. I urge you to investigate the websites <https://palletsprojects.com/p/flask/> and <https://palletsprojects.com/p/jinja/> to read more about these two powerful extensions to Python.

CHAPTER 6

Django

Django is a popular web development environment similar to Flask, but with some differences. The differences are significant in some circumstances. For instance, the Django APIs are a complete mechanism for managing all the parts including the database and template pieces. Flask, on the other hand, allows you to use the native API for these parts and thus gives you more choices for solving your problems. But if you are using pieces that Django supports, then this is the development environment you will want to use.

In this chapter, I will highlight the differences as they come up.

Django and WSGI

Django can be installed using `pip` or from your operating system repository. The `pip` command is shown here:

```
pip install django
```

This will install the latest version of Django. However, this may not be compatible with your web server or with your installed version of WSGI. Check before you install something so you don't have to replace it later.

It is highly recommended that, if possible, you install Django from your operating system distribution repository. This will ensure that the WSGI and Django versions will work together.

Django requires some changes to be made to the web server configuration file. The following are the changes that need to be added to the configuration:

```
WSGI PythonPath /pub/www/holmes4/mysite
WSGIScriptAlias /django /pub/www/holmes4/mysite/mysite/mysite.wsgi
<Directory "/pub/www/holmes4/mysite">
    Require all granted
</Directory>
```

The previous statements are similar to the statements required by Flask. The first statement points to the place where your Django modules are stored. Be careful here, because the `WSGI PythonPath` statement can go only in the web server's main configuration. It cannot appear in a virtual server configuration block. This means that if you are running multiple virtual servers, then all of them *must* share the same module directory. This can be a huge problem as it allows code to be shared across websites. So, be careful in your planning when using virtual web servers with Django.

The `WSGIScriptAlias` statement specifies the WSGI configuration file for your website. This file should never change in most cases. It tells WSGI the name of your application, which is usually `application`.

You should also note the `django` name added to the root of the script alias. What this name is does not matter. In fact, you may not even need a prefix name at all. We use one because we have both Flask and Django applications on our web server, and this prevents name collisions between the two environments.

The `Directory` block specifies the access permissions for the module directory. This is especially important if your module is outside the base web server directory of files.

Nothing needs to be installed on the web server for Django. Just make sure that your version of Django is compatible with the version of WSGI you are using.

Configuring and Testing Django

While you can use the Django virtual test environment to create your site's modules, you can just as easily create it by hand. If this is your first Django application, it is suggested that you use the Django virtual environment to create your first module. You can do this with the following command:

```
django-admin.py startproject mysite
```

You should run this from where you want to place this module. Do not worry too much about where you place it, as you will move it to its permanent location when you are done. When the command finishes, the following structure should be created in the current directory:

```
mysite
  manage.py
  mysite
    __init__.py
    asgi.py
    setting.py
    urls.py
```

The `mysite` directory names come from the last parameter on the `django-admin.py` command line. Feel free to change this to the name you want to use for your module.

The `manage.py` file is a command-line administration script for your module site. There are a number of things you can do with this script. Just type the script name without any parameters on the command line to discover the possible functions available.

The `asgi.py` file is the configuration file for WSGI. This file should not change except in extraordinary circumstances. In any case, when you deploy this module, you will need to rename the file. We renamed the file to `mysite.wsgi` when we deployed it for production. This name matches

what we used in the web server configuration file on the `WSGIScriptAlias` statement in the previous section.

The `settings.py` file has the settings to be used for this module. Some of these settings will need to be modified when you deploy the module to the web server.

The `urls.py` file is important. Every time you create a new function to be called from the browser command line, you will need to add a new line to this file. This file matches a name on the browser to a function to be called. This is different from Flask, which requires a `WSGIScriptAlias` statement for every function to be called in the module. This places that functionality inside a single Python file so that it does not clutter up the web server configuration file. And it prevents modifying the web server configuration file every time you want to add/delete/modify a function in the Django module.

To test our configuration, we need to build the classic “Hello World” function so it can be displayed by the browser. The first file we need is the Python file containing our callable function (Listing 6-1).

Listing 6-1. The `views.py` File

```
from django.http import HttpResponseRedirect

def hello(request):
    return(HttpResponse("Hello World!"))
```

Since this is just a test, our script is simple. We just want to make sure that a remote browser can connect and display the results of the hello function.

Next, we need to let the server know where the function resides. That is done in the `urls.py` file. The original contents of the file are displayed in Listing 6-2, and following it are our modifications.

Listing 6-2. The Original Contents of the `urls.py` File

```
"""mysite URL Configuration
```

The `urlpatterns` list routes URLs to views. For more information please see:

<https://docs.djangoproject.com/en/3.0/topics/http/urls/>

Examples:

Function views

1. Add an import: `from my_app import views`
2. Add a URL to `urlpatterns`: `path('', views.home, name='home')`

Class-based views

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `path('', Home.as_view(), name='home')`

Including another `URLconf`

1. Import the `include()` function: `from django.urls import include, path`
2. Add a URL to `urlpatterns`: `path('blog/', include('blog.urls'))`

```
"""
```

```
from django.contrib import admin
```

```
from django.urls import path, include
```

```
urlpatterns = [
    #path('admin/', admin.site.urls),
]
```

As you can see, the original file has no available routings to our `hello` function. The modification shown in Listing 6-3 adds the needed lines to make that happen.

Listing 6-3. The Modification Needed for the `urls.py` File

```
"""mysite URL Configuration
```

The `urlpatterns` list routes URLs to views. For more information please see:

<https://docs.djangoproject.com/en/3.0/topics/http/urls/>

Examples:

Function views

1. Add an import: `from my_app import views`
2. Add a URL to `urlpatterns`: `path('', views.home, name='home')`

Class-based views

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `path('', Home.as_view(), name='home')`

Including another `URLconf`

1. Import the `include()` function: `from django.urls import include, path`
2. Add a URL to `urlpatterns`: `path('blog/', include('blog.urls'))`

```
"""
```

```
from django.contrib import admin
from django.urls import path, include
from mysite.views import hello
```

```
urlpatterns = [
    #path('admin/', admin.site.urls),
    path('hello/', hello),
]
```

We have added two lines. The following line is needed so we can tell Python where our hello function is and how the URL for the browser is to be formed:

```
path('hello/', hello),
```

The first parameter to the path method is our URL modifier. The name here can be different from the function name, which is the second parameter. The URL for invoking the function will look something like the following example:

<http://www.holmes4.com/django/hello>

You should note how we have to add the django/ prefix we specified in the web server configuration. Again, we add this prefix to prevent name collisions between Django and Flask on our web server. The second name, hello, corresponds to the first parameter in the previous path function.

Just as important is the second modification to the `urls.py` file that we made.

```
from mysite.views import hello
```

This line instructs Python on the location of the hello function. Without this line, Python will not be able to locate the second parameter on the path function.

The next file we need to modify is the `settings.py` file. For this example, there are only two lines that need to be changed. The following line:

```
DEBUG = True
```

should be changed to this:

```
DEBUG = False
```

This will force the web server to present the normal error messages you would expect and log them in the `server_error_log`.

The next line:

```
ALLOWED_HOSTS = [ ]
```

should be modified so that it contains at least three entries in quotes separated by commas. The first entry should be the web server's IP address. The second entry should be the web server's DNS name. The third entry should be the remote client's IP address. You can also include an additional client IP address or a range of addresses.

At this point, you should restart your web server just to make sure that everything has been recorded by WSGI. Once that is done, enter the URL command shown earlier in your browser. If everything is correct, you should see the output in Figure 6-1 in your web browser window.

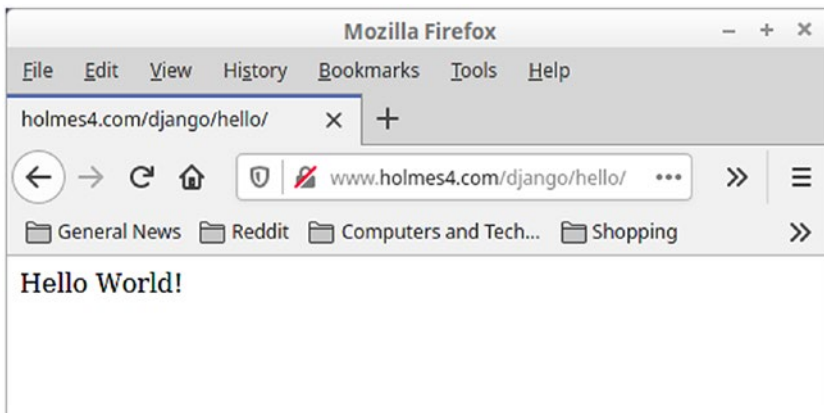


Figure 6-1. *The output from the hello command*

Congratulations, you have built your first Django application!

Django and Templates

Our next example Django application is the same as the Flask application we built. We are building this same application to show how applications built with Django are similar to those built with Flask. In fact, the templates in Django are similar to those in Flask.

To start, Listing 6-4 shows the additions to the `urls.py` file.

Listing 6-4. Additions to the `urls.py` File

```
"""mysite URL Configuration
```

The `urlpatterns` list routes URLs to views. For more information please see:

<https://docs.djangoproject.com/en/3.0/topics/http/urls/>

Examples:

Function views

1. Add an import: `from my_app import views`
2. Add a URL to `urlpatterns`: `path('', views.home, name='home')`

Class-based views

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `path('', Home.as_view(), name='home')`

Including another `URLconf`

1. Import the `include()` function: `from django.urls import include, path`
2. Add a URL to `urlpatterns`: `path('blog/', include('blog.urls'))`

```
"""
```

```
from django.contrib import admin
from django.urls import path, include
import mysite.views
```

```
urlpatterns = [
    #path('admin/', admin.site.urls),
    path('hello/', mysite.views.hello, name='hello'),
    path('hello/getdepartment/', mysite.views.getdepartment,
        name='getdepartment'),
]
```

There is only one line added to this file, and it supports the function `getdepartment()`. This points to the new function we will use to fetch a list of departments. The code for this function has been added to the `views.py` file; see Listing 6-5.

Listing 6-5. Additions to the `views.py` File

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import os, time

def hello(request):
    return(HttpResponse("Hello World!"))

def get_rows():
    rows = list()
    rows = ["DEPTNO DEPTNAME          MGRNO  ADMRDEPT
LOCATION                                ",
           "-----",
           "--",
           "A00   SPIFFY COMPUTER SERVICE DIV.   000010 A00   -",
           "B01   PLANNING                        000020 A00   -",
           "C01   INFORMATION CENTER              000030 A00   -",
           "D01   DEVELOPMENT CENTER              -      A00   -",
           "D11   MANUFACTURING SYSTEMS          000060 D01   -",
           "D21   ADMINISTRATION SYSTEMS         000070 D01   -",
```

```

        "E01    SUPPORT SERVICES          000050 A00    -",
        "E11    OPERATIONS                000090 E01    -",
        "E21    SOFTWARE SUPPORT          000100 E01    -",
        "F22    BRANCH OFFICE F2          -      E01    -",
        "G22    BRANCH OFFICE G2          -      E01    -",
        "H22    BRANCH OFFICE H2          -      E01    -",
        "I22    BRANCH OFFICE I2          -      E01    -",
        "J22    BRANCH OFFICE J2          -      E01    -",
        "",
        "  14 record(s) selected."
    ]
    return rows

def getdepartment(request):
    # create the substitutable content
    pagedict = dict()
    title = 'Table Display'
    title1 = 'Department Table'
    pagedict['title'] = 'Table Display'
    titleleft = 'Department Table'
    asctime = time.asctime()
    parts = asctime.split()
    year = parts[4]
    rows = get_rows()

    # get the template and make the substitutions
    t = get_template('getdepartment.html')
    html = t.render({'title': title, 'title1': title1,
                    'pagedict': pagedict,
                    'titleleft': titleleft, 'year': year,
                    'rows': rows})

    # return the substituted lines
    return HttpResponse(html)

```

As you can see, the code is similar to the Flask Python file that performs the same purpose. There is the function that returns the rows from the department we want to display plus code to set the variables containing the information we want to display. Finally, there is code to fetch the HTML template, substitute the variables into the template, and then return the finalized HTML. The main change here is that Django templates expect only a single Python dictionary rather than a list of values as separate parameters.

As for the template, you will find that at first glance it is exactly like the Flask template. While it is true the two template systems are similar, there are differences. The Django template appears as shown in Listing 6-6.

Listing 6-6. The Django Template for the `getdepartment()` Function

```
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="/css/
    example2-2.css" />
</head>
<body>
<!--Begin Header-->
<div id="header">
    <!-- Begin Picture Column -->
    <div id="header-left">
        <a href="/"></a>
        <br />image by W.David Ashley
    </div>
```

```

    <!-- Begin Title Column -->
    <div id="header-right"><h1>{{ title1 }}</h1>
    </div>
</div>
<!--Begin Page Menu-->
<div id="pagemenu">
    <!-- Begin Left Title Column -->
    <div id="pagemenu-left">{{ titleleft }}
    </div>
    <!-- Begin Page Menu Column -->
    <div id="pagemenu-right"><a href="/">&nbsp;Home&nbsp;</a>&nbsp;
    &nbsp;
        <a href="/">&nbsp;Menu2&nbsp;</a>&nbsp;
        <a href="/">&nbsp;Menu3&nbsp;</a>&nbsp;
    </div>
</div>
<!--Begin Content-->
<div id="content">
    <!-- Begin Content Menu Column -->
    <div id="content-left">
    </div>
    <!-- Begin Content Column -->
    <div id="content-right">
        <pre>{% for row in rows %}{{ row }}<br/>{%
        endfor %}</pre>
    </div>
<!-- Begin Footer -->
<div id="footer">
    <div id="footer-data">
        

```

```
        <br />&#169; Copyright 2010-{{ year }}  
        W. David Ashley. All rights reserved.  
    </div>  
</div>  
</body>  
</html>
```

Take a close look for the single difference in this template. It is in the line that prints out each row of the department data. The previous Flask template used `{{ row|e }}` to print each row of department data. The Django template eliminates part of that expression. The `|e` part of the statement used in the Flask template is a filter to help print an expression that has no characters associated with it. The Django template does not need this expression. This is just one of the small “gotcha” differences between the two template systems.

Just to prove the two systems can produce the same output, we will show you that output. See [Figure 6-2](#).

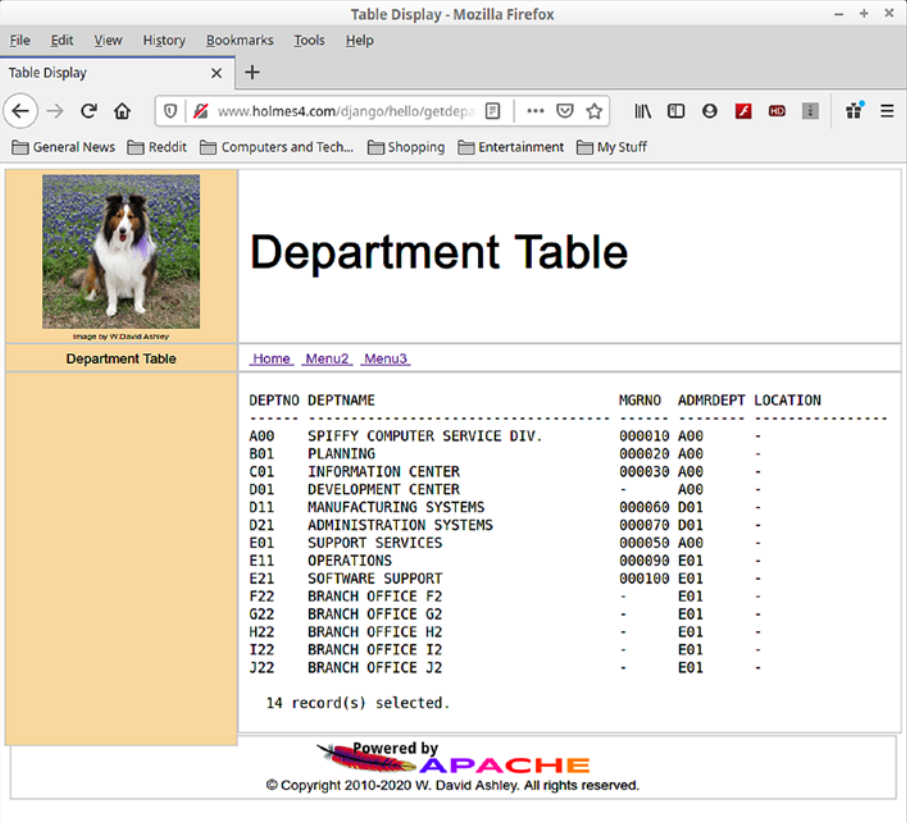


Table Display - Mozilla Firefox

Table Display

www.holmes4.com/django/hello/getdepa...

General News | Reddit | Computers and Tech... | Shopping | Entertainment | My Stuff

Image by W. David Ashley

Department Table

[Home](#) [Menu2](#) [Menu3](#)

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-
B01	PLANNING	000020	A00	-
C01	INFORMATION CENTER	000030	A00	-
D01	DEVELOPMENT CENTER	-	A00	-
D11	MANUFACTURING SYSTEMS	000060	D01	-
D21	ADMINISTRATION SYSTEMS	000070	D01	-
E01	SUPPORT SERVICES	000050	A00	-
E11	OPERATIONS	000090	E01	-
E21	SOFTWARE SUPPORT	000100	E01	-
F22	BRANCH OFFICE F2	-	E01	-
G22	BRANCH OFFICE G2	-	E01	-
H22	BRANCH OFFICE H2	-	E01	-
I22	BRANCH OFFICE I2	-	E01	-
J22	BRANCH OFFICE J2	-	E01	-

14 record(s) selected.

Powered by APACHE

© Copyright 2010-2020 W. David Ashley. All rights reserved.

Figure 6-2. Output from the `getdepartment()` function

The command we used was as follows:

<http://www.holmes4.com/django/hello/getdepartment/>

The output is the same for both the Flask and Django systems. Only the command is different. So, why use one system over the other? For that answer, please continue to the next chapter.

Using a Database with Django

This book is about dynamic web pages, but it is worth spending a little time with Django's support of multiple databases. Django supports a number of databases including SQLite, PostgreSQL, MySQL, Oracle, and Microsoft SQL in Windows. There are other packages available to support other databases such as IBM DB2. But for our purposes, it is not what databases are supported by Django; it is the way they are supported.

Much about database support in Django is surrounded by a library. It also assumes in a large way that you are working in concert with a database admin to design the database for your purposes. This also means that changes to the database design can be made during the implementation of the application you are working on. Unfortunately, this is not the way it works in the real world. In most situations, you will be stuck with a database design that cannot be changed because those changes will require other applications to accommodate those changes. This obviously will not happen in the real world.

So, what are your options? There are two obvious solutions: use the database design as is or create a new database that is a copy of the information in the real database. The second option is full of problems having to do with maintaining a copy of the data. Choosing the first option actually introduces a new set of options. You may choose to not utilize Django's database support at all and instead use a more native SQL support mechanism to access the database. The problem with this choice is that you will be leaving behind all the advantages that using Django's support will provide you and substituting a whole new code base that has separate support.

At this point, you may be asking yourself why you are using Django in the first place. If proper database access is a stumbling point for your application, then you may need to rethink your solution. Whether or not you choose Django as part of your solution, just make sure that your solution makes sense for your immediate needs as well as your future

needs. Careful planning on the front end will make a huge difference in the amount of work necessary in implementing your solution.

Summary

This chapter introduced Django as a dynamic HTML page mechanism. This mechanism can be used with or without a backend database. The template library used by Django is similar to the one used by Flask, so very little training will be necessary to switch from one template system to the other.

If you are going to use a database with Django, pay careful attention to the kind of database support Django offers as it may or may not be what you need in the short or long run.

CHAPTER 7

Comparing CGI, SSI, Flask, and Django

In this chapter, we will compare all the dynamic HTML page creation tools introduced so far. Each system has strengths and weaknesses and should be chosen with care. This will make it a lot easier to choose the system that fits your requirements.

In the discussions in this chapter, I will try to be as fair as possible. Obviously, I have my own preferences, but I will let the strengths of each system speak for themselves. When comparing the systems, do not let a single topic category sway your decision without also looking into the other categories. As much as you may like a single strength of a system, sometimes the weaknesses can far outweigh the downsides for your situation.

Installation and Configuration

In this section, I will discuss the installation and configuration for each dynamic HTML system. The advantages and problems we discovered will be discussed at length.

CGI Installation and Configuration

With most web servers, the CGI environment is installed by default. You only have to create your CGI scripts/programs, and you are off and running. There may be some configuration necessary to get your preferred script environment working under CGI, such as adding an `AddHandler` web server directive to enable your Python environment.

Of course, when using Python, the choice of a database is pretty wide open. The only caveat is that once you pick a database access method, you are pretty much stuck with it for the long term. You should carefully pick your database access method so that it will not become an issue later.

You can still use the method we discussed in [Chapter 2](#) to split the parts of your HTML pages into chunks that can be put together again dynamically. This should make it easier to create a multitude of HTML pages by putting together the different parts of your page without recoding it from scratch.

SSI Installation and Configuration

SSI is not usually configured by default in most web servers. You will need to add the proper configuration directives for SSI before it can be used. But once that is done, it opens up a wide variety of ways that SSI can be used. This includes adding dynamic text to your HTML pages and even calling your own scripts to completely create the entire page or just a piece of it.

Your database access method is still wide open when using SSI. The same precautions apply for SSI as for CGI programs when it comes to database access.

Splitting an HTML page is still possible when utilizing SSI and can offer some benefits without adding a performance hit to your pages.

Flask Installation and Configuration

Flask can usually be installed using either your operating system repository or the Python `pip` command. If available, installing it from the operating system repository is preferably so that it can be updated when the rest of the operating system is updated. Otherwise, you will be stuck manually ensuring that your version of Python is compatible with the version of Flask you are using.

Flask will also require a fair amount of configuration to enable its use with your web server. But once you have your first script working, subsequent scripts will be much easier to configure.

Your database access method is also wide open when using Flask. Just be sure that your choice will not have an adverse impact in the future.

While it is possible to split your HTML pages into multiple pieces under Flask, it will be a little harder to submit that page to the Jinja2 system for variable substitution. It can be done, but you will need to look up the process for managing that submission.

Django Installation and Configuration

Django can also be installed using either an operating system repository or the Python `pip` command. If the option is available, installing it from the operating system repository is preferable so that it can be updated when the rest of the operating system is updated. Otherwise, you will be stuck with manually ensuring that your version of Python is compatible with the version of Django you are using.

Django also requires a fair amount of configuration to enable its use with your web server. But once you have your first script working, subsequent scripts will be much easier to configure.

Database access with Django is somewhat limited to what Django can support directly. You must also have permission from your database admin to modify the database as needed by Django. This will be easy if you are setting up a new database, but much harder if you will be accessing an existing database.

As far as splitting your HTML pages into different parts, it will be a little harder to submit those pages for processing by Django's page parser. But it can be done if you are careful.

Python Usage

All of the products we have utilized in this book use the Python language. But, does a system use Python to its advantage, or does the language get in the way of accomplishing your goal? This section will examine this aspect to see how well Python is used in each product.

CGI Python Usage

We made some use of Python in CGI scripts to break apart HTML skeletons and then put them back together again to make the best use of the HTML and CSS frameworks. This allowed us to create customized HTML pages for use in different parts of our website for different purposes. The ease of putting the HTML parts back together with Python allowed for very dynamic page designs with little effort on our part.

Another useful feature was using Python's library to create small dynamic portions for the HTML page (like the copyright year) that allowed the page to dynamically change without any update to the page by an editor. This kept the number of updates to a page to a reasonable number and allowed us to concentrate on the page information content and not the minutiae.

SSI Python Usage

SSI can be a strange beast to work with. On the one hand, it supplies some nice capabilities; on the other, it limits us in how we can put an HTML page together. Python does not help much with this because using it to deliver the page itself eliminates the usage of any SSI capabilities. However, SSI can give us one capability not available to CGI. When a Python script is invoked via the standard CGI mechanism, the current directory is always the CGI directory. By using SSI to invoke a CGI script, the script inherits the directory where the HTML/SSI page was originally located. This can be useful if the script needs to utilize the contents of that directory to display a list of files. But the new HTML page will lose the ability to be parsed by SSI.

However, you may still find a limited use for SSI, and I encourage you to make use of it where feasible to do so.

Flask Python Usage

Flask makes use of Python to a great extent—so much so that your script is usually just setting some variables that will later be used by the HTML page template processor. This tends to make your main Python program small and efficient. The one drawback is that a Flask program is like a CGI program in that the script always has the same current directory inherited from the Flask environment. Thus, a Flask script cannot make use of any of the files in the web server's file system.

It also can be a little harder to break apart your HTML files and dynamically put them back together again before submitting them to the template processor. This task, while not impossible, is harder to accomplish because all the page parts must be located within the Flask file system. This may or may not mean that your parts library is inside the web server's file system. It will depend on how you have located the Flask environment directory.

Django Python Usage

Django also make great use of Python. It is similar to Flask in that most of your script will be setting Python variables for use later by the Django template processor. Django is also similar to Flask in that your scripts will always have the same current directory, which is inherited from the Django environment. Thus, Django cannot make use of any files in the web server's file system.

The same limitation is also true for breaking apart your HTML pages and dynamically putting them back together again. This task, while not impossible, is harder to accomplish because all the page parts must be located within the Django file system.

Template Processing

Perhaps the feature most used when dynamically producing HTML pages is some sort of template processor. There are actually a number of these facilities available in Python, and any of them can be used. It is even possible to use Django's template system with Flask, and vice versa. Template systems tend to just be Python modules, so you could install the system of your choice and use it for your HTML output.

CGI Template Processing

The CGI system was born long before the first Python template processor. Thus, most existing CGI systems process their own HTML dynamically via the CGI script. But if CGI is your choice for a dynamic HTML environment, there is nothing to stop you from using a modern Python template system to help you create dynamic HTML pages. Just make sure your choice of a template processor matches your system requirements.

SSI Template Processing

In a word, *stop*. It is not possible to use a template system if you have SSI directives in your HTML page. The SSI directives will be ignored because the SSI processor will be bypassed.

Flask Template Processing

Flask has a powerful template processor. It has lots of features and capabilities. In fact, there are so many features that you will probably never utilize even half of them. Just keep in mind that if you run across a formatting problem, you should first check out the template system before you use Python or create HTML to solve the problem.

Django Template Processing

The Django template processor is slightly different from the Flask processor, but it has more similarities than differences, and it is usually not too hard for someone to switch from one system to the other. Just as with Flask, keep in mind that if you run across a formatting problem, you should first check out the template system before you use Python or create HTML to solve the problem.

Database Access

While not strictly covered in this book, this subject is worth covering here because it can have a profound impact on your problem solution. In fact, most dynamic HTML systems are created strictly to process database actions. So if you choose the wrong database access method, the impact can be severe in both problem solution development and database access response.

One thing to always remember is that a database can hold many types of data, be organized differently than others of the same data, and may require specialized methods to obtain appropriate response times. It may take some experimenting to find an access method that has the capabilities and provides response times suitable for your situation. Do not be afraid to put in some up-front time on this choice because it can save you a lot of time on later during the development process by eliminating surprises.

Rest APIs

Django has one advantage not offered by any of the other systems: Rest APIs. The Django Rest Framework (DRF) is used extensively throughout the industry to increase the efficiency and response to the database. The framework is easy to use and quite efficient at decreasing the response time for queries to the database system. Of course, these can be added to any system we have discussed, but the built-in APIs in Django offer an easy-to-use framework automatically available with Django.

Summary

Each system we discussed has great features and some drawbacks. Most of the drawbacks have to do with whether a feature can solve your problem. If Django does not support your database, then there are strong reasons to select another system. Whatever system you choose will have some features that are hard to distinguish from a feature in another system, such as the Django and Flask template systems.

It will be worth your time to choose a system carefully. It will pay big dividends in the end.

Index

A, B

- Apache directives, 27
- Apache Tomcat server, 6
- Apache web server
 - /cgi-bin subdirectory, 24
 - CGI directory, 3
 - config file, 3
 - DocumentRoot
 - directive, 22
 - htaccess/htpasswd files, 23
 - httpd.conf file, 6–8, 10, 11, 13–15, 17, 19
 - Include directive, 20
 - Keep-Alive loop, 5
 - Listen directive, 20
 - mod_mime_magic
 - module, 25
 - Options directive, 22
 - request-response loop, 3–5
 - sendfile syscall, 25
 - ServerAdmin directive, 21
 - ServerRoot directive, 20
 - sprintf format, 24
 - user directive, 20
 - <VirtualHost> containers, 21

C

- Cascading Style Sheets (CSS), 2, 29
 - border/background
 - properties, 43–45
 - box model, 39
 - box model properties, 46, 47
 - other properties, 50
 - parent-child box model, 39
 - selectors, 38, 40–43
 - text properties, 47, 48
 - transition, animation and
 - transform properties, 49
- cgi-bin directory, 71, 100
- cgi-bin program, 71, 73, 75
- CGI program
 - calling data, 100
 - GET method, 101, 103–106
 - header HTML part, 76, 78
 - portable/maintainable, 78–80, 82, 84
 - POST method (*see* POST method)
 - strategy, 75
- Common Gateway Interface (CGI)
 - definition, 1
 - installation/configuration, 202

INDEX

Common Gateway

- Interface (CGI) (*cont.*)

- Python usage, 204

- template processing, 206

Conditional directives

- binary operators, 153

- endif, 147

- expr attribute, 149

- functions, 156

- HTTP page, 147

- if directives, 157, 158

- output, 148

- QUERY_STRING, 149

- unary operators, 154, 155

- variables, 150–152

- config SSI directive, 125, 126, 128

- content_right element, 83

Cookies, CGI

- key-value pairs, 119, 120

- sending, HTTP server, 117, 119

- stateless protocol, 117

D

- Database access, 207

- department() function, 177

Django

- asgi.py file, 185

- configuration, 184

- databases, 198

- Directory block, 184

- error_log, 189

- getdepartment() function, 194, 195, 197

- hello function, 189

- installation/configuration, 183, 203

- manage.py file, 185

- module creation, 185

- mysite directory, 185

- output, 190

- Python usage, 206

- settings.py file, 186, 189

- structure, 185

- template processing, 207

- urls.py file, 186–189, 191, 192

- views.py file, 186, 192, 193

- virtual environment, 185

- WSGI PythonPath

- statement, 184

- WSGIScriptAlias

- statement, 184, 186

- Django Rest Framework (DRF), 208

- Document/metadata tags, 31

E

- echo SSI directive, 129, 130

- envvar.py program, 75

- exec SSI directive, 130, 132–134

F

Flask

- @app.route() decorator, 169

- configuration, 167

- DOCUMENT_ROOT, 170

- flasktest.py file, 168, 169

- flasktest.wsgi file, 167
- installation/configuration, 166, 203
- invocations, 169
- libraries, 167
- output, 174
- Python usage, 205
- sudo command, 166
- sys.path.insert() statement, 168
- template processing, 207
- flastmod SSI directive, 138, 140
- fsize SSI directive, 135, 137

G

- Generalized Markup Language (GML), 29
- get_rows() function, 177
- Grouping content tags, 33, 34

H

- html_lib directory, 84
- HTML tags
 - creating forms, 36, 37
 - creating tables, 35, 36
 - CSS, 30
 - document/metadata, 31
 - embedding content, 37, 38
 - GML, 29
 - grouping content, 33, 34
 - section, 34, 35
 - select Hyperlink, 94, 96, 98–100
 - SGML, 30

- text, 32, 33
- Hyperlink skeletons, 84–94
- Hypertext Markup Language (HTML), 2, 29
 - creating library, 65–67
 - CSS pages, 52, 53, 55, 57, 60, 62, 63, 65
 - page footer, 68
 - website directories, 51
- Hypertext Transport Protocol (HTTP), 2

I

- include SSI directive, 140, 142
- Installation/configuration
 - CGI, 202
 - Django, 203
 - Flask, 203
 - SSI, 202

J, K, L

- Jinja2
 - database query, 175, 176
 - decorator statements, 172
 - and Flask, 172
 - framework, 171
 - gotchas, 180
 - HTML statement, 173
 - HTML templates, 177, 178
 - looping expression, 179
 - output, 174, 179, 180
 - render_template, 172

INDEX

Jinja2 (*cont.*)

- statements, [174](#)
- string module, [171](#)
- template, [173](#)

M

Modules, [3](#)

N

Nginx web server, [5](#)

O

Operating system links, [26](#)

P, Q

POST method

- check forms, [107–110](#)
- crlf characters, [113](#)
- dropdown box, [115, 117](#)
- HTML forms, [107](#)
- TextArea, [112](#)

Python usage

- CGI, [204](#)
- Django, [206](#)
- Flask, [205](#)
- SSI, [205](#)

R

Request-response cycle, [3](#)

Rest APIs, [208](#)

S

Sectioning tags, [34, 35](#)

Server Side Includes (SSI)

- conditional directive
(*see* Conditional directive)
- configuration
 - statements, [123](#)
- installation/configuration, [202](#)
- Python usage, [205](#)
- scripting language, [123](#)
- set directive (*see* Set directive)
- template processing, [207](#)
- XBitHack directive, [124, 125](#)

Set directive

- curly brackets, [143](#)
- HTML, [144, 145](#)
- output, [146](#)
- variable names, [143](#)
- var/value attributes, [142](#)

.shtml file extensions, [124](#)

Standardized General Markup

Language (SGML), [30](#)

Stateless protocol, [117](#)

sys.path.insert() function, [168](#)

T, U, V

Template processing

- CGI, [206](#)
- Django, [207](#)
- Flask, [207](#)
- SSI, [207](#)

Text tags, [32](#)

W, X, Y, Z

Web Server Gateway Interface

(WSGI), [159](#), [183](#)

application, [164](#)

configuration, [161](#)

HTML, [165](#)

installation, [160](#)

mod_wsgi, [161](#)

output, [165](#), [166](#)

wsgitest.wsgi script file, [162–164](#)