FOUNDATIONS OF DEBUGGING FOR GOLANG

MATTHEW BOYLE

byteSizeGo

CONTRIBUTORS

ABOUT THE AUTHOR

Matt Boyle is an experienced technical leader in the field of distributed systems, specializing in using Go.

He has worked at huge companies such as Cloudflare and General Electric, as well as exciting high-growth startups such as Curve and Crowdcube.

Matt has been writing Go for production since 2018 and often shares blog posts and fun trivia about Go over on Twitter (@MattJamesBoyle[1]).

ABOUT THE TECHNICAL REVIEWERS

Ansar Smagulov is a full-stack engineering lead specializing in Go and TypeScript.

Ansar has led teams and projects at Central Asia's largest cloud provider, PS Cloud Services. He also develops Zen, a free and efficient privacy guard for desktop operating systems, among other open-source projects.

You can find more about Ansar's open source work at GitHub (@anfragment[2]).

Toni Lovejoy is an experienced software engineer who has been writing production Go code for the last 5 years. With much of her experience in high growth startups such as StockX.

She has given back to the community teaching backend coding at bootcamps and volunteering her time teaching coding to youth.

You can reach Toni on Linkedin[3].

Swastik Baranwal is an open-source developer. He has been contributing to The V Programming Language[4] as one of the core developers and has greatly impacted the open-source community and projects.

His interest lies in system programming and continuously learning new technologies. He is open to working full-time as well as part-time roles.

To learn more about him, you can visit his website https://swastik.is-a.dev/

Michael Bang is a pragmatic developer that loves to spend his time building maintainable and well-tested software.

He fell in love with Go more than a decade ago and has used it happily ever since. He loves helping the community in small ways from the shadows; **Foundations of Debugging for Golang** is the fourth book he has contributed to as a technical reviewer. If you need help with a technical book, you're welcome to reach out to @micvbang[5].

CONTENTS

METHODS OF DEBUGGING

DEBUGGING IN PRODUCTION

WELCOME!

Welcome to this book, **Foundations of Debugging for Golang.**

I believe debugging is probably the most important skill that most people are never taught.

Being able to **debug** locally and in production is a **critical skill** for any Go engineer, but it is rarely taught explicitly. It has taken me many years of working with Go in production to get comfortable debugging, and **I want to accelerate your learning by teaching you everything I wish I had known when I started out.**

This book started life as a course and was turned into a book by popular request. If you are a junior or mid-level Go engineer, I think this book will be useful for you.

I have written this book from the ground up, but the content still overlaps with the course somewhat. If you are more of a visual learner like me, and as a thank you for buying this book, you can get 20% off the course from https://bytesizego.com using the code **BOOK20**.

When people think of debugging, they often think of the debugger. We *will* cover how to use the debugger in this book, but it is a small part of our debugging utility belt. We will also cover:

- How to get better at spotting issues by eye.
- Logging patterns.
- Metrics.
- Distributed Tracing.
- Profiling & Pprof.

If you don't know what any of those things are, do not worry, as by the end of this book you will.

I hope you enjoy this book and I look forward to hearing what you think! The best way to reach me is over on Twitter (or X): @matt-jamesboyle. [1]

- Matt Boyle

WHAT IS DEBUGGING AND WHY DO WE DO IT?

Debugging is an essential part of programming. It refers to the process of identifying, analyzing, and fixing issues in code. The term "**debugging**" originates from a fascinating story involving computer pioneer Grace Hopper.

Whilst working on the Harvard Mark II computer in the 1940s, Hopper's team encountered a system malfunction. Upon investigation, they discovered a literal bug - a moth stuck in one of the machine's mechanical relays! The bug was causing the relay to fail, leading to errors. Hopper's team removed the moth, taped it to their logbook, and coined the term "**debugging**" to describe fixing the glitch.

byteSizeGo

When debugging code ourselves, it helps to think of Grace Hopper's moth and methodically follow discrete steps:

Firstly, we need to identify the bug

This can be tricky, as some only appear in specific situations like high load.

Having robust logging and monitoring helps detect bugs early. Furthermore, writing comprehensive tests also surfaces issues before they impact users. We will not talk about writing tests in this book, but we will discuss how to debug them.

Secondly, we analyze the problem to determine the best fix

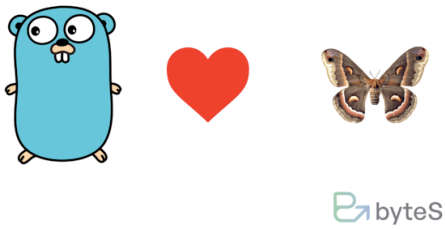
Should we add a quick if statement as a patch? Should we do a bigger refactor? Each approach has its merits. The former tackles the immediate issue. However, considering the long-term perspective, the refactor is likely a better solution.

If the bug is surfaced in an ongoing incident, I would often recommend making the quickest fix you can for now to remove customer impact (after ensuring we have captured the information we need to follow up later). We can make a ticket on the backlog for the bigger refactor.

Finally, we implement the fix and update associated tests.

This ensures the bug stays fixed and the new code maintains integrity.

Debugging techniques range from carefully reading code to advanced tooling. Thankfully, Go provides excellent built-in tools. From simple print statement debugging to robust profilers and tracers, Go makes finding and fixing bugs easy.



Learning to debug will make you a better software engineer. Mastering the various debugging techniques will enable you to identify and resolve issues more efficiently. Debugging is challenging but rewarding. With practice, you will build skills to create reliable, robust applications.

Just like Grace Hopper debugging the Harvard Mark II, we too can methodically squash the bugs in our code!

METHODS OF DEBUGGING

In this section we will start by learning some high level techniques for getting better at debugging (such as how to get better at spotting them by eye or pairing with a buddy). After that, we will dive into logging and discuss strategies for how to ensure you don't log too much; a common problem I see.

We'll finish the chapter out by learning about the debugger, ensuring you can get it setup and we'll learn some more advanced techniques for using it.

I hope you enjoy it and learn a bunch!

DEBUGGING BY EYE

As a Go developer, one of the most fundamental skills you'll need is the ability to debug by eye. This involves developing an instinct to spot issues just by looking at code.

Whilst tooling has come a long way, and Go tooling really is excellent, it's important to build this fundamental skill too.

There are no shortcuts to getting better at debugging by eye. Espe-

cially when starting out, try and stick with an issue for a while; it is how you will grow as a developer.

In the new world of AI tools, it can be tempting to paste an issue into ChatGPT for an instant fix, and many developers do this regularly (including me). However, try not to rely too heavily on AI assistance whilst learning a new language or concept, unless you are completely stuck. This is because when you try ten ways to fix an issue and nine of them don't work, you actually just learnt ten things. As an example, I remember spending a lot of time early in my Go career trying to figure out why my program was terminating without running any logic. I tried lots of different ways to fix it, and learnt about signals, channels and how goroutines works; things I still use all time. Using AI can help you accelerate this learning if you are disciplined enough to ask it to explain the solution it gave you - but be sure to always double check it too - hallucinations are very real!

Debugging by eye may sound simple, but it's a really powerful tool. Over time, you'll develop a sixth sense for spotting anomalies and potential bugs just by reading your code. It's not just about finding errors; it's about understanding the flow and logic of your program.

Remember, Go's compiler catches many types of issues, but not everything.

A SIMPLE EXERCISE

Let's start with a simple exercise to train your eye to catch a real issue. Take a moment to look at the following piece of Go code and see if you can spot the bug:

```
package main

import "fmt"
```

```
func main() {
    numbers := []int{1, 2, 3, 4, 5}
    fmt.Println("Sum:", sum(numbers))
}

func sum(numbers []int) int {
   total := 0
    for i := 0; i <= len(numbers);  i++ {
       total +=  numbers[i]
    }
    return total
}
```

Can you spot the issue? It's a really small change. The change is underlined on the next page.

```
package main

import "fmt"

func main() {
    numbers := []int{1, 2, 3, 4, 5}
    fmt.Println("Sum:", sum(numbers))
}

func sum(numbers []int) int {
   total := 0
    for i := 0; i <= len(numbers); i++ {
       total += numbers[i]
    }
    return total
}
```

With the original code, we're ranging over a slice of integers. Our loop invariant states that we will loop less than or equal to the length of the slice. In Go, slices are zero-indexed, and therefore, we would receive an index out-of-range error when we run this code. This would not be a compile-time error; it would be a runtime error, meaning our program will crash when we run it.

This may not seem like a big deal in this situation, but if this piece of code did not have unit tests and was deployed to production, it could lead to a customer-facing issue.

STRATEGIES FOR EFFECTIVE CODE INSPECTION

There are several strategies to make your code inspection more effective:

• Learning code patterns.

• Being aware of common errors.

• Pair programming.

We'll discuss each of these strategies in the following sections to help you become more proficient at debugging by eye.

LEARNING CODE PATTERNS AND BEING AWARE OF COMMON ERRORS

As a Go developer, being able to read and understand code patterns is an essential skill. Go, like any language, has its own unique idioms and constructs that you'll encounter time and time again. Familiarizing yourself with these patterns will not only help you comprehend and debug Go code more effectively but also enable you to write more idiomatic and maintainable code.

ERROR HANDLING

Unlike in many other languages, in Go, errors are not exceptions but values that can (and should!) be propagated and handled explicitly. Understanding the idiomatic way of handling errors in Go is crucial to writing robust and reliable code. Here's a simple example:

```
func doSomething(flag bool) (bool, error) {
    if !flag {
        return false, errors.New("something_wen-
t_wrong")
    }
    // If everything is fine, return a re-
sult and nil as the error
    return true, nil
}
```

We can then return this error to the caller function who can handle it explicitly.

INTERFACES

Interfaces in Go are a way of defining behavior, and they play a central role in achieving abstraction and code reusability. Being able to recognize and work with interfaces is a fundamental skill for any Go developer.

Learning interfaces is beyond the scope of this book. If you feel you have not quite grasped them yet, I recommend the *gobyexample interface tutorial*[1] .

CONCURRENCY

Concurrency is another core feature of Go, so I recommend getting familiar with goroutines and channels. These patterns can be tricky to master, but once you do, they'll open up new possibilities for writing efficient and scalable code. The official Go site has a nice interactive introduction to concurrency[2].

STYLEGUIDES

Reading and understanding style guides can be a great way to get a sense of what idiomatic Go code looks like. The Go team[3] and organizations like Uber[4] have published their style guides, which can serve as excellent resources for learning best practices and avoiding common pitfalls.

In addition to style guides, there are tools like Golangci-lint[5] that can help you catch common code smells and potential issues. These tools can be invaluable for catching subtle bugs and improving the overall quality of your code.

As you gain more experience with Go, you'll start to develop an eye for spotting potential issues in code. This skill, often referred to as "debugging by eye," can save you a lot of time and effort in the long run.

ANOTHER EXERCISE

Below is another exercise to practice debugging by eye. It's followed immediately by the solution so try not to skip ahead! See if you can spot the issue, simply by reading the code carefully.

```
package main

import (
    "fmt"
    “os”
    "strings"
)

func main() {
    content, _ := os.ReadFile("config.txt")
    if strings.Contains(string(con-
tent), "enable_feature") {
        fmt.Println("Feature Enabled")
    } else {
        fmt.Println("Feature Disabled")
    }
}
```

Solution

Here, we're ignoring the error returned by os.ReadFile by assigning it to a blank identifier (_). This can be problematic because if the file cannot be read (for example, due to permissions issues or the file not existing), the program will execute the wrong logical path, potentially leading to unexpected behavior or even crashes.

WRAPPING UP

This has been a gentle introduction to the concept of debugging. It may seem simple, but this fundamental skill - debugging by eye - can save your bacon in many cases.

If you want further practice, try to identify and fix the bugs in the following code snippets without running the code. This exercise will help you apply what you've learned and sharpen your debugging abilities.

Take a look at the code below and try and fix all the issues. By my count, there are four.

You can also see this code on the Go Playground here [6].

```
package main

import (
    "fmt"
    "strconv"
)

type Test struct{
    str *string
}

func main() {
    a,_ := strconv.Atoi("thirty three")

    var numbers = []int{1, 2, 3}
    for i := 0; i <= len(numbers); i++ {
        fmt.Println(numbers[i])
    }

    t := Test{}
    if t.str == string(a) {
        fmt.Println("should log")
    }
}
```

Can you spot them all? If not, grab a buddy and head on over to the next chapter on pair programming to learn how you can work together to debug. If you don't have anybody to pair with right now, don't worry - you can watch me tackle the solution here[7].

PAIR PROGRAMMING

Pair Programming is one of the best ways to grow as a Go engineer, especially if you are fortunate enough to pair with someone with more experience than yourself. As the more senior member of a pair, there is still plenty of benefit to you too - we will explore that more below.

When I used to work 100% in the office, we used to pair often and I credit it with much of my technical growth. I found moving to remote

working has made pairing much harder, but I found a few ways to make it work (although I think the experience is still inferior). I'll share these with you at the end of the chapter.

WHAT IS PAIR PROGRAMMING?

Pair programming involves two programmers working together on the same piece of code, collaborating in real time. It's a dynamic process that fosters knowledge sharing, problem-solving, and effective communication – skills that are essential for any successful software engineer but can often be overlooked.

THE DRIVER AND THE NAVIGATOR

In pair programming, there are two distinct roles: the driver and the navigator. The driver is the one with their hands on the keyboard, writing the code and making the changes. The navigator, on the other hand, acts as a coach, guiding the driver, reviewing the code as it's written, and serving as a *rubber duck* for the driver to bounce ideas off.

SWITCHING ROLES

To get the full benefits of pair programming, it's important to switch roles regularly; I recommend every 45 minutes or so (and that you pair for at least for an hour and a half). This role reversal ensures that both programmers have the opportunity to experience the different perspectives and challenges associated with each role, ultimately leading to a more well-rounded understanding of the codebase and the problem at hand.

THE POWER OF TWO MINDS

One of the most significant advantages of pair programming is the collective brainpower it brings to the table. When two developers collaborate, they can catch mistakes, identify potential issues, and

explore alternative solutions more effectively than when working alone. This is particularly valuable when debugging tricky issues or tackling complex problems that have left you stumped.

ACCELERATING LEARNING AND KNOWLEDGE TRANSFER

Pair programming is a powerful learning tool, especially when it involves pairing a junior developer with a more experienced one. Juniors get to learn from seniors, absorbing their knowledge, coding patterns, and debugging strategies. But the learning process goes both ways – seniors can also gain fresh perspectives and insights from their junior colleagues. For example, once you have worked somewhere for a long time, you become used to the way of working there - you might not mean to but you're starting to let yourself slip into the "we have always done it this way" mentality. Someone with a fresher perspective can challenge your assumptions which will potentially lead to a better outcome.

Pair programming also facilitates the exchange of valuable tips, tricks, and productivity hacks that might otherwise go unnoticed. From discovering a colleague's favorite terminal or a handy script that streamlines their workflow, these small gems can significantly enhance your overall development experience. I actually learnt to use the debugger this way.

BUILDING RAPPORT AND FOSTERING COMMUNICATION

Beyond the technical benefits, pair programming also offers a unique opportunity to build rapport and strengthen communication skills with your colleagues. Working closely with someone, exchanging ideas, and collaborating on a shared goal can foster a sense of camaraderie and teamwork that is bigger than any feature you can deliver.

Effective communication is a crucial skill for any software engineer, and pair programming provides a practical environment to hone this skill. By verbalizing your thought processes and ideas to your partner,

you'll develop the ability to explain complex concepts clearly and concisely which will help you throughout your career.

TO PAIR OR NOT TO PAIR?

Whilst the frequency and approach to pair programming may vary across companies and teams, here are some times where I believe it to be valuable.

Debugging Tricky Issues

When you've spent hours staring at a problem and can't seem to spot the issue, a fresh set of eyes can do wonders.

Tackling Complex Problems

Pair programming can be invaluable when faced with tricky challenges that require diverse perspectives.

Onboarding new Team Members

Pairing a new hire with an experienced developer can accelerate their learning and help them get familiar with the codebase and team practices.

Regular Knowledge Sharing

Incorporating pair programming sessions into your team's routine can foster continuous learning and knowledge transfer among team members.

TIPS FOR BEING A GREAT DRIVER

1. **Stay Focused:** As the driver, your primary responsibility is to write the code. Stay focused on the task at hand and avoid distractions. Ensure that your coding is clear and understandable, as this will make it easier for your navigator to follow along and provide input.

2. **Communicate Clearly:** Keep an open line of communication with your navigator. Explain your thought process as you code, explain why you are making certain decisions, and ask for feedback regularly. This helps maintain a collaborative atmosphere and ensures that both of you are on the same page.
3. **Be Receptive to Feedback:** The navigator is there to help guide you and catch potential mistakes. Be open to their suggestions and critiques. Remember that constructive feedback is a crucial part of the learning process and will help you become a better coder. This does not mean you need to accept every suggestion though!
4. **Think Aloud:** Verbalizing your thought process can help the navigator understand your approach and provide more effective guidance. It also allows for real-time problem-solving and idea sharing, enhancing the overall collaborative experience.
5. **Maintain a Steady Pace:** Avoid rushing through the code. Maintain a steady pace that allows both you and your navigator to think through each step thoroughly. This ensures higher code quality and a better learning experience for both parties.
6. **Write Clean and Readable Code:** Strive to write code that is clean, well-documented, and adheres to best practices. This not only helps the current session but also ensures that the code remains maintainable for future reference.
7. **Stay Organized:** Keep your work environment organized. Close unnecessary windows and tabs and keep your workspace tidy. This minimizes distractions and helps you stay focused on the task at hand.
8. **Ask for Help When Needed:** Don't hesitate to ask your navigator for help if you're stuck or unsure about something. Pair programming is a collaborative effort, and seeking assistance when needed is a sign of effective teamwork, not a weakness.

9. **Review Code Regularly:** Periodically review the code with your navigator to ensure that it meets the desired standards and functions as intended. This practice helps catch potential issues early and promotes a continuous feedback loop.
10. **Stay Patient and Positive:** Pair programming can sometimes be challenging, especially when dealing with complex problems. Stay patient, keep a positive attitude, and remember that the goal is to learn and grow together.

TIPS FOR BEING A GREAT NAVIGATOR

1. **Stay Engaged:** As the navigator, it's crucial to stay actively engaged throughout the session. Pay close attention to the driver's actions, provide timely feedback, and be ready to offer suggestions or corrections when needed.
2. **Guide, Don't Dictate:** Offer guidance and advice without taking over the driver's role. Suggest improvements and point out potential issues, but allow the driver to make the final decisions and type the code. This helps foster a collaborative environment.
3. **Ask Questions:** Encourage the driver to explain their thought process by asking clarifying questions. This not only helps you understand their approach but also prompts them to think critically about their decisions.
4. **Provide Constructive Feedback:** Offer feedback that is specific, actionable, and focused on improving the code and the driver's skills. Frame your comments in a positive and supportive manner to foster a collaborative atmosphere.
5. **Think Ahead:** Anticipate potential problems and think about the bigger picture. While the driver focuses on the immediate task, you should consider the long-term implications, possible edge cases, and overall design of the code.
6. **Share Knowledge:** Use the opportunity to share your expertise and thoughts. Provide tips, tricks, and best practices

that the driver might not be aware of. This not only helps improve the current code but also contributes to the driver's professional growth.

7. **Stay Patient and Respectful:** Be patient, especially if the driver is less experienced or takes longer to understand certain concepts. Respect their pace and learning process. Remember that the goal is to support and help them grow.
8. **Balance Guidance and Silence:** Know when to speak up and when to stay silent. Too much input can be overwhelming, while too little can leave the driver feeling unsupported. Strike a balance by providing guidance at key moments and allowing the driver space to think and work independently.
9. **Encourage Experimentation:** Support the driver in exploring different approaches and solutions. Encourage them to try out new ideas, even if they might not work out. This fosters a learning environment and can lead to innovative solutions.
10. **Maintain a Positive Attitude:** Keep the mood positive. Pair programming can be intense, and maintaining a positive attitude helps keep the session productive and enjoyable. Celebrate successes and view mistakes as learning opportunities.

PAIRING REMOTELY

Pairing in person is fairly easy; you can pass the keyboard back and forth and take breaks to grab a coffee and catch up about Love Island. Doing it remotely takes discipline and patience. There is going to be times when your partner's internet is flaky, they have to answer the door or the cat decides to sit on the keyboard; this is normal and should be embraced and used as an opportunity to have a quick break or non-code based chat. Using pairing as an opportunity to build friendship and rapport is encouraged!

There is a whole host of tools out there to "help" you pair remotely; some of them are free and some of them are quite expensive. In most cases, I still just share my screen via Zoom or Google Hangouts and this is sufficient. I have tried some of the tools created for remote pairing such as Pair With Me[1]from JetBrains and, whilst very technically impressive, I found it can actually detract from the process as each participant gets their own cursor and view of the code - it is therefore very easy to get distracted and go and look at a different file or jump ahead.

My advice here: keep it simple to start with. If you pair with someone a lot and want to explore some of the fancier tools then give it a try. Let me know if you come across a great one!

HOW I HAVE USED PAIR PROGRAMMING TO SOLVE PRODUCTION ISSUES

It all started with a service that seemed simple enough. We were initializing some error groups in the main.go file for a set of dependencies. The code would loop over these groups and fire off a function – in this case, making HTTP requests. Everything was compiling without issue, and the logic appeared sound. However, as is often the case, appearances can be deceiving. It looked something like this:

```
package main

import (
    "fmt"
    "golang.org/x/sync/errgroup"
)

func main() {
    var g errgroup.Group
    values := []int{1, 2, 3}
    for _, v := range values {
        g.Go(func() error {
            if _, err := process(v); err != nil {
                return err
            }
            return nil
        })
    }

    if err := g.Wait(); err != nil {
        fmt.Println("Encountered error:", err)
    } else {
        fmt.Println("No error encountered")
    }
}

// Dummy function to simulate some processing
func process(n int) (int, error) {
    fmt.Println(n)
    if n > 1 {
        return 0, fmt.Errorf("num-
ber too high: %d", n)
    }
    return n, nil
}
```

To better understand the problem, I added a log to the process function. I clicked "Run" and examined the output. To my surprise, the values being logged was 3, 3, 3, when I expected it to be 1,2,3.

In situations like these, the temptation to reach for a debugger or rely heavily on logging can be strong. However, I've learned that sometimes the only way to truly understand an issue is to collaborate with a colleague, research the underlying mechanics of the language, and walk through the code line by line.

After some investigation and pair programming, we stumbled upon the missing piece of the puzzle: a single line of code that made all the difference. By introducing ***v := v*** before launching the goroutine, the output transformed from 3, 3, 3 to the expected 1, 2, 3 (albeit in a different order, as is typical with goroutines). Here's a snippet of the updated code.

```
for _, v := range values {
      v := v
       g.Go(func() error {
           if _, err := process(v); err != nil {
               return err
           }
           return nil
       })
```

The root cause lies in a subtlety of Go's behavior: variables using goroutines are evaluated when the routine is executed, not when it's launched. Without the additional line of code, all goroutines were referencing the same variable, which had been updated to the last value of the loop by the time they executed. This gotcha has since been addressed in Go 1.22, but there is still plenty of code out there using 1.21 and below. It is also a great example of why being a Software Engineer requires you to be a lifelong learner.

This experience taught me a valuable lesson: sometimes, the most effective debugging technique is to slow down, examine the code with a friend and lean on their experience.

Whilst modern tools have their place, there's no substitute for good old-fashioned code comprehension and collaboration. And who knows? The solution to your next production puzzle might just lie in the simple act of "debugging by eye."

LOGGING

Logging is an essential part of the software development process, and usually the first debugging technique folks learn.

In this chapter, we'll dive into the world of logging and logging strategies. We're going to start by talking about logging generally before moving on to structured logs and how they differ from the unstructured logs you might be used to.

Structured logs are more than just plain text; they're organized, machine-readable data structures that can be easily parsed and

analyzed. We'll explore the benefits of structured logging and how it can streamline your debugging processes, making it easier to identify patterns, correlate events, and extract valuable insights from your logs.

By the end of this chapter, you'll have a solid understanding of logging's role in your technology stack and how it can contribute to your company's maturity journey. Whether you're a seasoned developer or just starting out, this chapter will equip you with the knowledge and skills necessary to use logging effectively, ensuring that you can tackle debugging challenges with confidence and efficiency.

Let's get to it!

LOGGING LOCALLY WITH THE FMT PACKAGE

Go has several ways to log, the simplest being the `fmt` package. Most Go engineers when starting out will learn about this package pretty early and it's a great place to start your logging journey.

Let's go through some of the most common functions you might want to use.

fmt.Print

This function prints an argument to the standard output. It's straightforward to use:

```
fmt.Print("hello")
```

Which would simply output `hello.`

Even for complex data types like structs, fmt.Print handles the formatting in an easy to consume format. Consider the following code:

```
type user struct {
          name string
          age  int
}

u := user{
          name: “Matt",
          age:  3,
}
fmt.Print(u)
```

Will print **{matt 3}.** Simple, but effective.

fmt.Println

This function is similar to **fmt.Print** but adds a new line character at the end, ensuring each print statement appears on a new line. This can make the output more readable, especially when printing multiple lines.

```
type user struct {
          name string
          age  int
}

u := user{
          name: “Matt",
          age:  3,
}
fmt.Println(u)
fmt.Println(u)
```

Would print:

```
{Matt 3}
{Matt 3}
```

As you can see, each output it printed on a new line

`fmt.Printf`

This function allows you to format the output using placeholders. It's one of the most commonly used printing functions, offering flexibility and readability.

```
  type user struct {
        name string
        age  int
 }

u := user{
        name: “Matt",
        age:  3,
}
  fmt.Printf(“The user’s name is %s”, u.name)
```

Will print **`The user’s name is Matt.`**

There are different placeholders for different types.[1] Here we used **`%s`** which is the placeholder for strings, but I also commonly use **`%v`** (for printing structs), and **`%d`** (for printing integers).

The **`fmt`** package will take you a surprisingly long way. Whilst you’re just starting out, you should embrace it. However, it is missing a couple of features that can make our logs a little more useful. Let’s take a look at another standard library package that can make our logs a little more awesome.

UPGRADING LOGS WITH THE LOG PACKAGE

While the **`fmt`** package is great for quick and simple logging, the log package offers more robust and configurable logging capabilities. It provides features like automatic timestamping, severity levels, and the ability to write logs to different output streams. We will not spend too

much time talking about the log package, as newer versions of Go include another package (slog) that I think is even better than this one, especially for production. This is because slog has more powerful and expressive API that allows us to embed data into the logs in a cleaner way. However, for local logging, the log package is a decent middle ground.

The log package defines a **`Logger`** type with methods for formatting output, as well as helper functions for common logging tasks. It also includes additional functions like **`Fatal`** and **`Panic`**, which can be used to handle critical errors and exceptional situations.

Let's look at our basic example from above again:

```
log.Print("hello")
```

Outputs:

```
2024/03/31 Hello
```

This automatic timestamp can prove incredibly useful, especially if our logs are going to be ingested into another system, to ensure order is preserved.

The API for the log package is very similar to that of fmt. All of the below will work:

```
  type user struct {
        name string
        age  int
 }

u := user{
        name: "Matt",
        age:  3,
}
  log.Print("Hello")
```

```
    log.Println("Hello")
    log.Printf("The user's name is %s", u.name)
```

As mentioned, the package also has **log.Fatal**:

```
log.Fatal("oops")
log.Print("Hello")
```

The above code will print `oops` but not `hello`. This is because log.Fatal logs a message and then calls **`os.Exit`**, terminating the program. This function should be used sparingly, and typically only in the main function, when the program encounters a critical error and cannot continue executing.

Similarly, the **`log.Panic`** function logs a message and then panics, which I recommend you almost never use. This is because panicking in code is the equivalent of an "uncontrolled explosion". Your application will shut down and you do not get the ability to shut down any dependencies - for example closing a database connection.

One of the advantages of using the log package is its customizability. It allows you to control the output format, including timestamps, file paths, and line numbers. However, for simple local debugging, the default settings may be sufficient. The log package provides a straightforward way to get started with logging without the need for extensive configuration.

Whilst the log package is a great tool for local debugging, it may not be the best choice for production environments, for the reasons we outlined above. The slog package offers advanced features like structured logging, which can provide better organization and filtering capabilities for your logs. We'll look at that in the next section.

SO WHICH SHOULD I USE FOR LOCAL LOGGING?

If you're working on a small project or simply need to quickly debug an issue locally, the fmt package is likely the better choice. Its simplicity and ease of use make it a great starting point for developers of all skill levels.

Once you are ready to move to production, I recommend slog.

SLOG

In Go 1.21, slog was introduced to the standard library[2]. Before this, structured logging had to either be built manually or by using a third party library (I typically used Zap[3] from Uber).

The slog package is a structured levelled logger and enables us to turn logs that look like this using the log package:

```
2009/11/10 23:00:00 hello, world{matt 30}
```

Into the following using slog:

```
2009/11/10 23:00:00 INFO hello, world impact-
ed_user="{user:matt age:30}"
```

As you can see we have gained the **`INFO`** keyword and the ability to add attributes that have printed in a **`JSON`** format. This is structured, leveled logging and is how I recommend you log for production.

Structured Logging

The problem with logs generally is they are unstructured. This means they do not adhere to a consistent, predefined format, making them hard to query and sort, which are two traits that are pretty critical if we intend for our logs to be ingested into another system so we can use them for debugging.

For log files to be human readable, we commonly structure them in JSON format. Slog is our means to do that easily in Go. We saw a basic example of info logging with slog previously, but you may have noted that the output was still not in JSON format. To do that we need to do something like the following:

```
logger := slog.New(
    slog.NewJSONHandler(
        os.Stdout,
        nil
    ),
)

logger.Info("hello, world",
"user", "Matt",
)
```

Which will produce:

```
{"time":"2023-08-04T16:58:02.939245411-
04:00","level":"INFO","msg":"hello,
world","user”:”Matt”}
```

We chose to write to **`os.Stdout`** here but you could also write directly to a file or another system too.

We saw previously that we could attach key/value pairs to logs such as a struct or a just a user's name. This is useful, but would get annoying if you wanted to do it on every log. You can use **`LogAttrs`** to attach key value pairs to every log instead. This would look as follows:

```
slog.LogAttrs(
    context.Background(),
    slog.LevelInfo,
    "hello, world",
```

```
    slog.String("user", "Matt")
)
```

As you can see, **LogAttrs** takes a context. This means a handler can access this and pull out values such as the trace ID (we talk about tracing later in this book).

Here's a slightly more complex and real-world example. Consider a scenario where you are building a web application and want to include user-specific information in your logs for every request. Without **slog.LogAttrs**, you would need to manually add the user information to each log entry, which is repetitive and error-prone. With **slog.LogAttrs**, you can set these attributes once and have them automatically included in all subsequent logs.

```
func handleRequest(ctx context.Con-
text, userID string) {
    ctx = context.WithVal-
ue(ctx, "userID", userID)

    slog.LogAttrs(
        ctx,
        slog.LevelInfo,
        "Processing request",
        slog.String("userID", userID)
    )

    slog.LogAttrs(
        ctx,
        slog.LevelInfo,
        "Request processed successfully",
        slog.String("userID", userID)
    )
}
```

There is much more to slog, but beyond log levels, you now know near enough everything you need to know to start producing structured logs!

Log Levels

Log levels are a way to distinguish between different types of log messages based on their importance or severity. The slog package supports several log levels out of the box, including Info, Warning, Error, and Debug. By default, slog outputs all logs at the Info level and above.

You can adjust the log level dynamically using the **`slog.Level`** option. For example, to set the log level to Error and above, you can use the following code:

```
opts := &slog.HandlerOptions{
        Level:  slog.LevelError,
    }

    logger := slog.New(slog.NewJSONHandler(os-
.Stdout, opts))

    err := errors.New("some-error")

    logger.Info(
        "Hello World",
        slog.String("meta_info", "some-
thing else"),
        slog.Int("account_id", 35464),
        slog.Any("err", err),
    )

    logger.Error(
        "Hello World",
```

```
        slog.String("meta_info", "some-
thing else"),
        slog.Int("account_id", 35464),
        slog.Any("err", err),
    )
```

This code would only output:

```
{"time":"2009-11-10T23:00:00Z","level":"ER-
ROR","msg":"Hello World","meta_info":"something
else","account_id":35464,"err":"some-error"}
```

The info log would be completely ignored because the log level has been set to error which is a higher severity than info. For completeness, here is all the log levels provided by the slog package, in order of priority:

```
Debug < Info < Warning < Error < Critical
```

CREATING A LOGGING STRATEGY

I recommend keeping things simple and only having two log levels in your application, `Info` and `Error`.

By default, your log level should be set to `Error`, but you can enable `Info` by switching an environment variable. This stops your logging systems from becoming overwhelmed with noise generally, but does enable you to "turn it up" for periods of time if you need to, without having to make code changes and redeploy your application.

Generally, logging should happen at the "edge" of your application. This means I tend to only log in **`main()`**, http handlers or gRPC service implementations. Following this approach centralizes error handing, helps to reduce noise, and forces you to think about user impact. To do this successfully, you will need to use **`fmt.Errorf`** to

"wrap" errors further down your stack to bubble them up to the layer in which we can log them. Here's an example:

```
// readConfig simulates reading a configura-
tion file.
func readConfig(path string) (*config, error) {
    _, err := os.ReadFile(path)
    if err != nil {
        return nil, fmt.Errorf(
            "failed_to_read_config_file: %w",
            err
        )
    }
    return &config{}, nil
}

// initApp initializes the application and re-
turns an error if it fails.
func initApp() error {
    _, err := readConfig("config.txt")
    if err != nil {
        return fmt.Errorf(
            "initialization failed: %w",
            err
        )
    }
    return nil
}

func main() {
    logger := slog.New(
        slog.NewJSONHandler(
            os.Stdout,
            nil
        )
```

```
    )
    err := initApp()
    if err != nil {
        logger.Fatal(
            "App startup failed",
            slog.Error("err", err)
        )
    }
    logger.Info("App started successfully")
}
```

The above also showcases two different log levels happening. We have a call to **`Fatal`**, which will log then terminate our application. We would likely be always interested in seeing these in standard out, or pushing them to our logging tool. However, the info log might be something we enable if we are debugging, but not something we need to see always.

Deciding if a log should be info or error is not a perfect science. My advice is to use error log if something is *actually* an error; for example saving to the Database failed, a call to a Stripe failed, a file you tried to read didn't exist. If you are unsure, start with an info log and upgrade it.

WHAT SHOULD I DO WITH ALL THESE LOGS?

I have mentioned a few times in this chapter about logging systems and using your logs to debug, but how do you actually do that if they are all just going to standard out?

The simplest thing you can do is output logs to a file. You then have the ability to review these during or after an event to piece together what is happening. This is very challenging and does not scale very well.

A common next step is to push your logs into something like Elastic-Search[4] and view them with Kibana[5]. In the exercise below I have included a docker-compose file and a means to export your logs and view them in Kibana, so check it out and give it a go!

Once your logs are in something like Kibana, you have the ability to run queries against them, build dashboards and even setup alerts if you see an increase in a certain amount of logs within a time period.

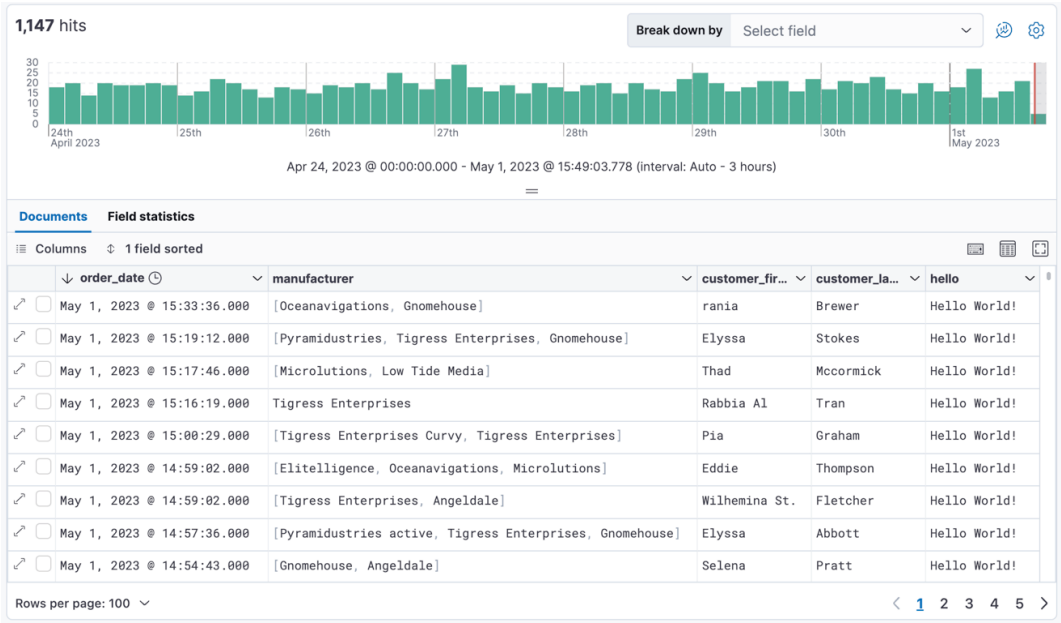


Image courtesy of elastic.co

Having either Kibana or similar setup is, in my opinion, the minimum that should be done before running a system in production. If you are serving customers and don't have this level of observability setup, I urge you to spend some time next week making it so. There are both great cloud and open-source options so there is no excuse!

AN EXERCISE

This exercise is longer than the previous ones, but as part of *the Ultimate Guide to Debugging with Go* course [6] I teach, there is an excuse which has you submit logs to a Kibana instance we setup using

docker. This is a great learning experience in my opinion, so I wanted to include it in the book too.

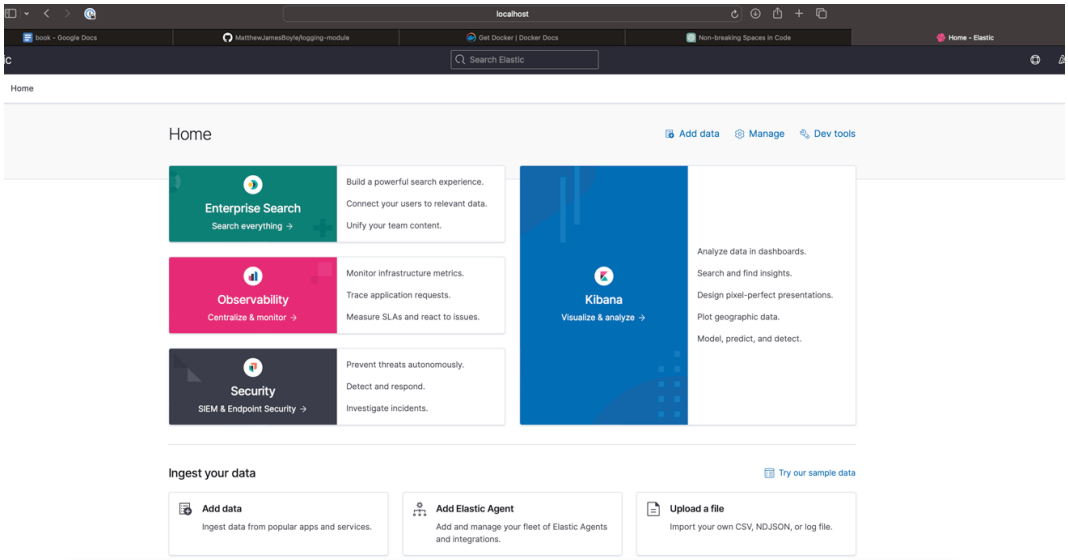
The goal of this exercise is to add your own endpoint to a Go project I have created, and ensure it has structured logs too.

This exercise is split into a few parts.

Firstly, clone the repo here: https://github.com/MatthewJamesBoyle/logging-module/tree/main. Once cloned, run ***`docker compose up.`*** You'll need to have docker installed for this to work, which you can get here[7]. When you run the command, you'll see lots of output in your console. When you see:

```
logging-module-kibana-1  | {"type":"log","@timestamp":"2024-05-24T10:03:18Z","tags":
["listening","info"],"pid":9,"message":"Server running at http://0:5601"}
logging-module-kibana-1  | {"type":"log","@timestamp":"2024-05-24T10:03:19Z","tags":
["info","http","server","Kibana"],"pid":9,"message":"http server running at http://0:5601"}
```

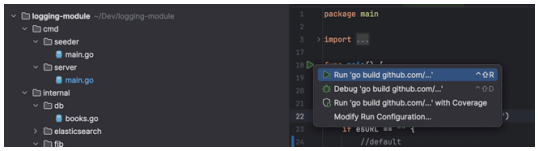
Things are running and you're good to go! Kibana shcomprehending system behavior ould now be available to you at http://localhost:5601 in your browser. The first time you open it, it should look something like this:



You don't need to do anything else here yet.

Elasticsearch will be available at http://localhost:9200. This does not have a UI, and this is where will submit logs to.

In your IDE, you should now navigate to **cmd/server/main.go** and run the application. In Goland, I do this by clicking the green play button like this and it's similar in VScode.

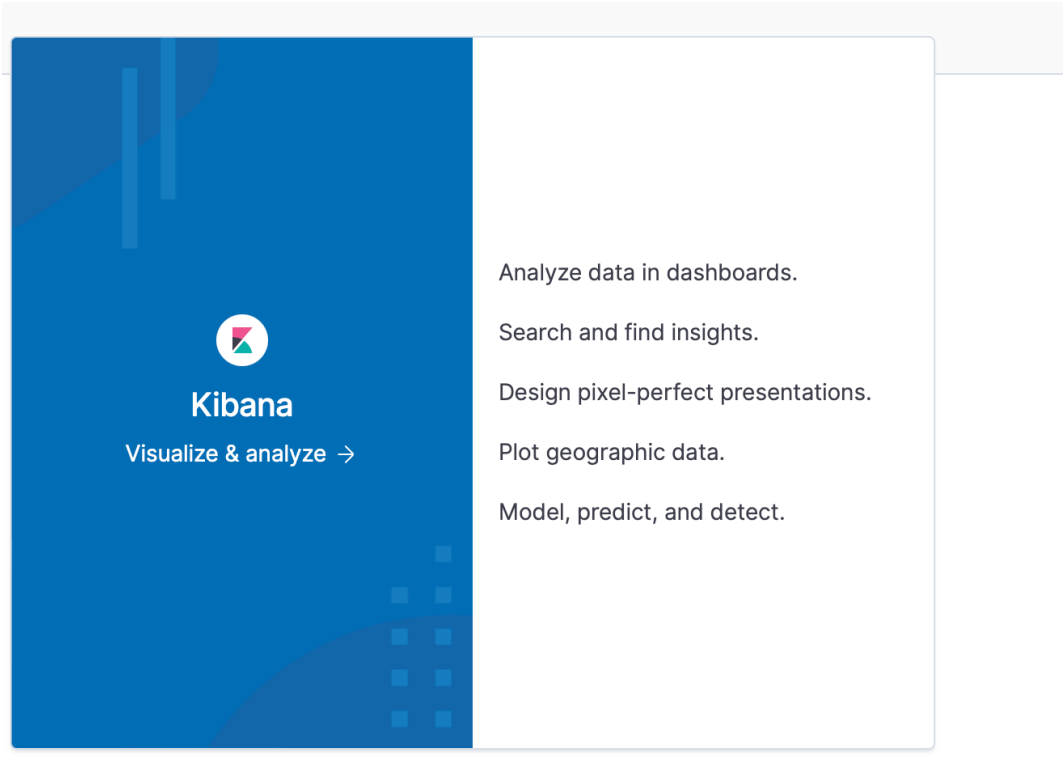


Once running, make some cURL requests by issuing the following command in a terminal:

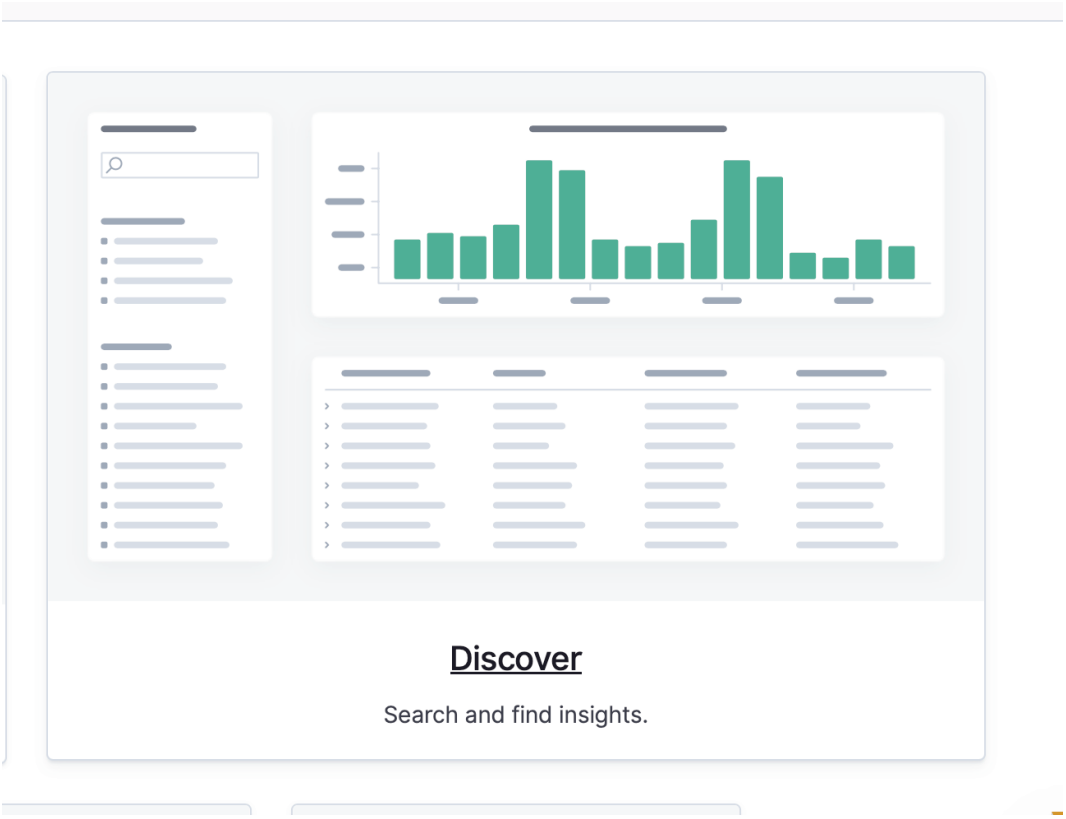
```
curl --location 'http://localhost:8080/books'
```

If you do it a few times, you'll see various success and failures, such as I did below.

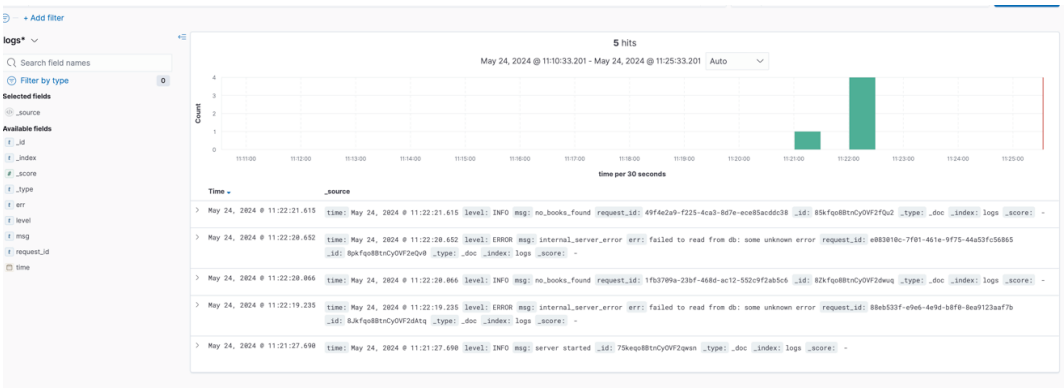
If you now go back to Kibana, click "Visualize & analyze" :



And then click “Discover”



You should see some logs from our service there. It will look something like this:



Have a play around here. Make some more requests and experiment with the filters. Most places I have worked use Kibana somewhere in their stack, so experience with it is a great skill to have.

Now you have everything running, take a look at the **`GetBookByAuthor`** function. The exercise is to create an endpoint such as:

```
/book?author=$authorName
```

Using the code in the service. To do this you'll need to add some code to **`transporthttp.go`** and **`dbadaptor.go`**, adding logs as you go. Once this is done, review your endpoint and ensure you are happy with the logs.

Final part. A customer has got in touch and said they keep receiving an error "author not supported", even though they are making a request for "rachel barnes" which the library says should be a supported author. Why might this be? Can you debug it using only logs in Kibana?

Here is the request the system is making for that user:

```
curl --location 'http://localhost:8080/book?
author=Rachel%20Barnes'
```

A guided walk through of this exercise can be found in the course here[8].

WRAPPING UP

I did not expect to write so much about logs at any point in my life, and I bet you never expected to read so much either, yet here we both are!

I hope you have a good understanding of how to approach logging in your next Go project. Before we move on from logging, I want to provide a couple of warnings for completeness sake. The below is the equivalent of the disclaimer that shows up before you download software that you're probably not going to read, but I wouldn't sleep at night if I didn't share it!

- Excessive logging can introduce significant overhead to your system's performance. Each log operation consumes CPU cycles and I/O bandwidth.
- High volumes of logging can increase response times, impacting user experience, especially in high-transaction systems (This is why toggling between log levels is a good idea in my opinion).
- Logging too much data can lead to rapid consumption of storage space, leading to increased costs and management overhead.
- Managing, archiving, and purging large volumes of logs can become a challenging and resource-intensive task.
- Overlogging may inadvertently include sensitive information, posing a risk to data privacy and violating compliance

standards. A larger volume of logs increases the risk of exposing critical information in the event of a security breach.
- Too much log data can obscure important information, making it harder to identify critical issues amidst the noise.
- More logging means more storage and processing power, leading to higher infrastructure costs.
- The cost of managing and maintaining large log systems, including staff time and tools, can be high.

Phew! That's a lot to take in. Remember, logging is a powerful tool that, when used effectively, can greatly enhance your ability to monitor and debug your applications. By considering these points, you can ensure that your logging strategy is both effective and efficient.

Throughout this chapter, I have shared my views on how to manage all of the above, but I really recommend you read this x thread[9] when some folks have a different view than me. I always advise reading widely and considering different opinions!

THE DEBUGGER

It's hard to write a book on debugging without mentioning the debugger.

In this chapter, we will ensure that you have the debugger setup in either VSCode or Goland, and then learn how to use everything from the beginner to the advanced features. All the setup instructions below are for Mac, but I don't think they should differ too much by

Operating System. If you do have any issues here, email me on hello@bytesizego.com and we'll get you setup.

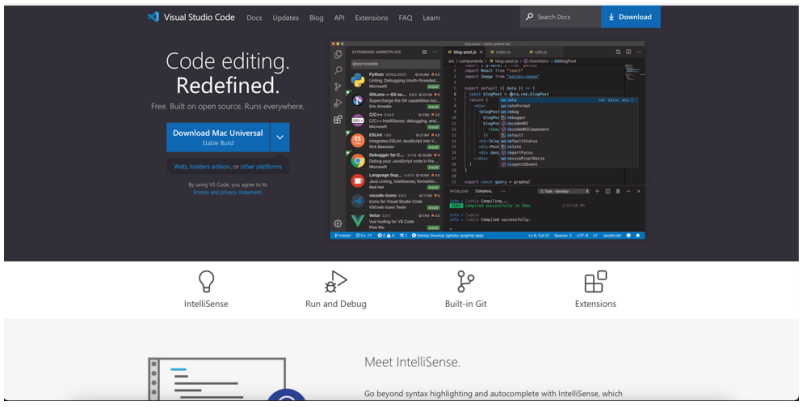
This chapter will probably not be that useful to you unless you actively have an IDE in front of you, so I recommend coming back to this chapter later if you're currently on the Northern Line to Edgware.

If your choice IDE is not VSCode or Goland, that's fine. Everything I discuss in this chapter should be possible with any Go IDE, but the workflow might vary slightly. If you run into any issues, drop me an email and I'll help you figure it out.

If you don't have an IDE installed or no preference, I have a strong preference for Goland as I think the developer experience is vastly superior, but it does cost money. However, there is a 30 day trial so you can try it out[1]. VSCode is completely free but does require further customization and setup.

SETTING UP THE DEBUGGER IN VSCODE

Firstly, navigate to https://code.visualstudio.com/ [2]and download the installer for your machine. Once downloaded, open the installer and follow the prompts to install VS Code.

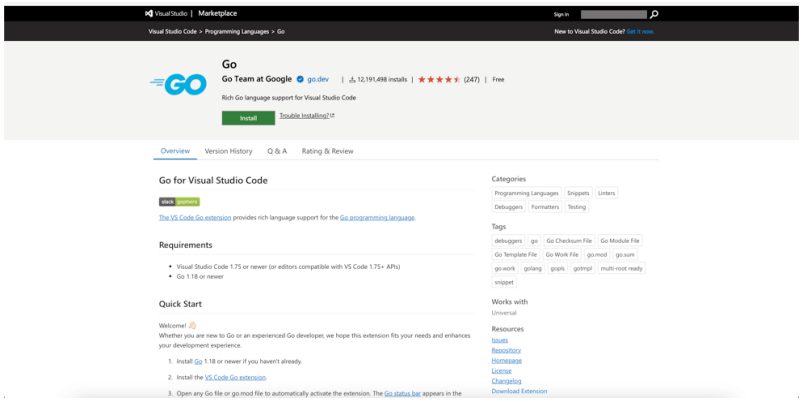


Next, open a Go project. Any will do. Creating a new one is fine also.

You can open an existing project by selecting "Open Folder" from the welcome screen or by going to File > Open Folder.

To enable debugging features for Go, you'll need to install the Go extension from the VS Code Marketplace using this link:

https://marketplace.visualstudio.com/items?itemName=golang.go[3].

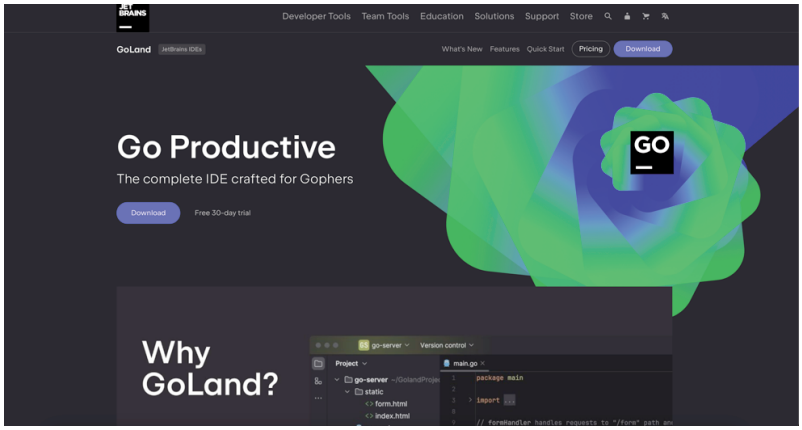


Once you have done this, you might notice a banner in the bottom right corner of VSCode, with a message saying some other Go dependencies are missing or there are missing pieces or tools. If you click on it, it'll tell you that you need to install **Go PLS** and some other things to make the debugger work. Just click on it and hit install.

You should be good to go with VSCode!

SETTING UP THE DEBUGGER IN GOLAND

To get setup with Goland is even easier; simply download and install Goland from here: https://www.jetbrains.com/go/.



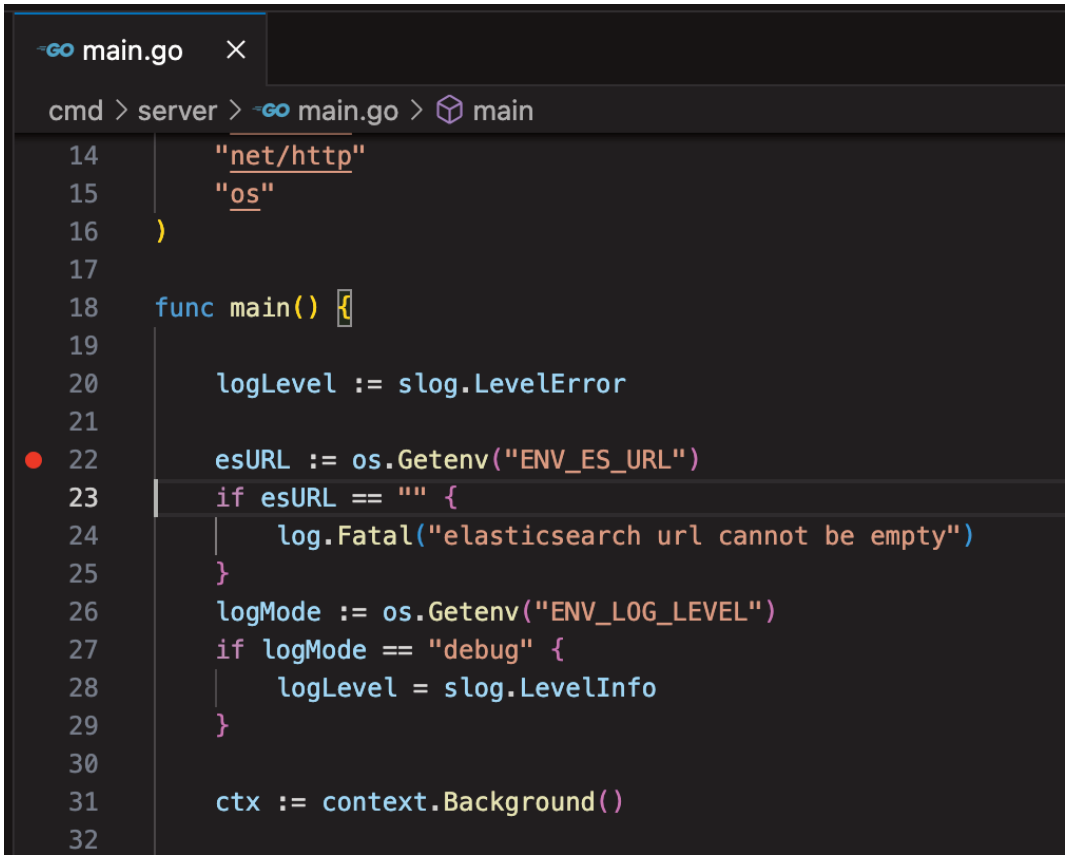
Once installed, you are good to go.

BREAKPOINTS

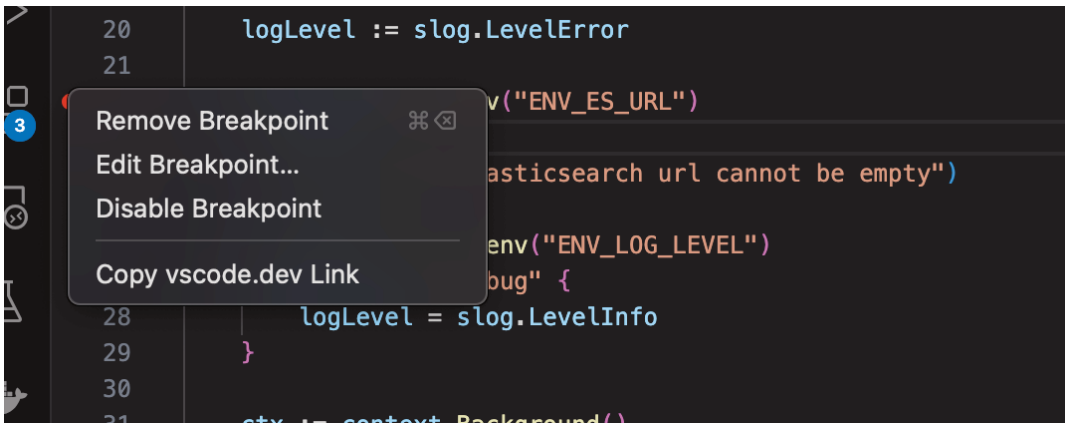
Breakpoints can be thought of as a pause in your program.

If there is a specific line, value or point in our program where slowing things down to allow further analysis would be helpful, we can set a breakpoint to be able to do it.

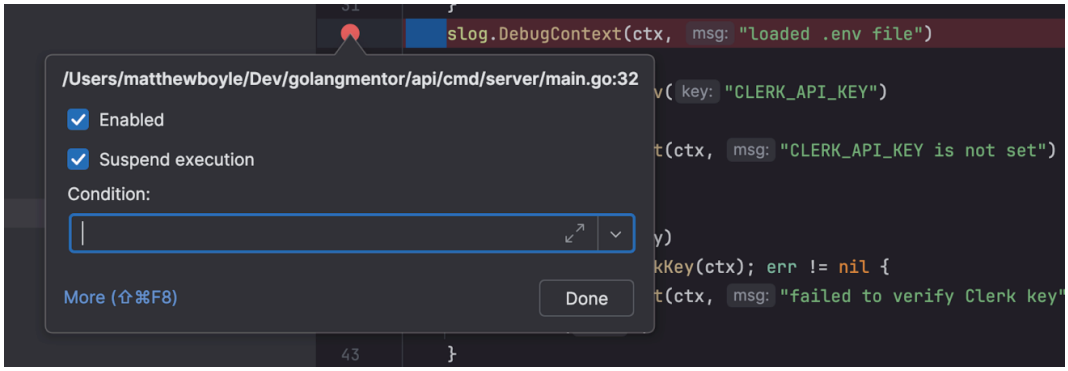
To set a breakpoint in both VSCode and Goland, we simply need to left click in the gutter (the left hand side where the line numbers are) and it will put a small dot there to acknowledge the fact that we would like the program to pause there.



In VScode, If you want to remove or disable a breakpoint, you can simply right click on it and you'll see the following options:



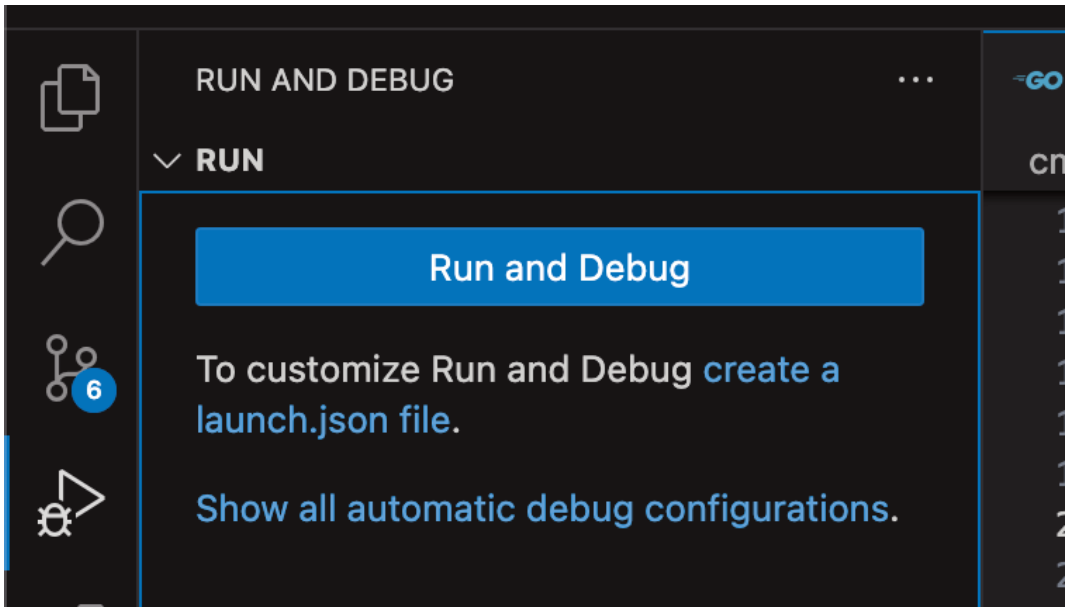
In Goland, you can simply left click the breakpoint again to remove it. You can also right click it to suspend it or add conditional logic:



We'll talk about conditional breakpoints later in this chapter.

Once we have a breakpoint set, it is time to run the debugger.

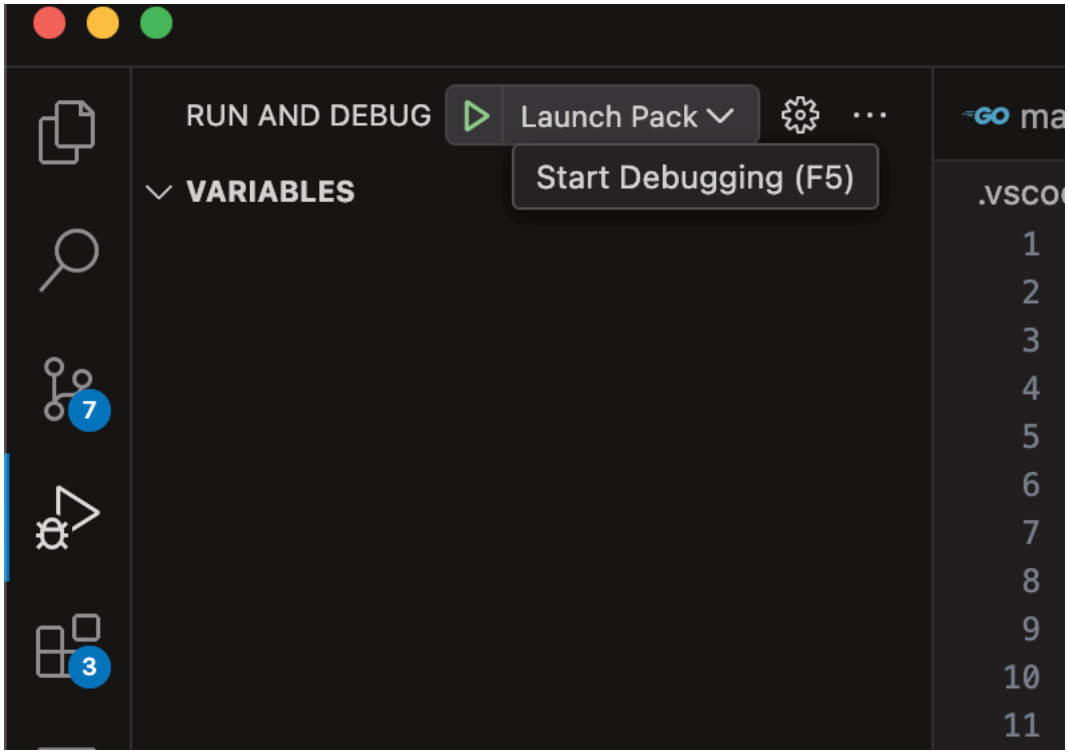
In VSCode, you can click the "Run And Debug" button on the left hand side (bottom icon in the screenshot provided). The first time you do it you will see that you need to create a launch.json file.



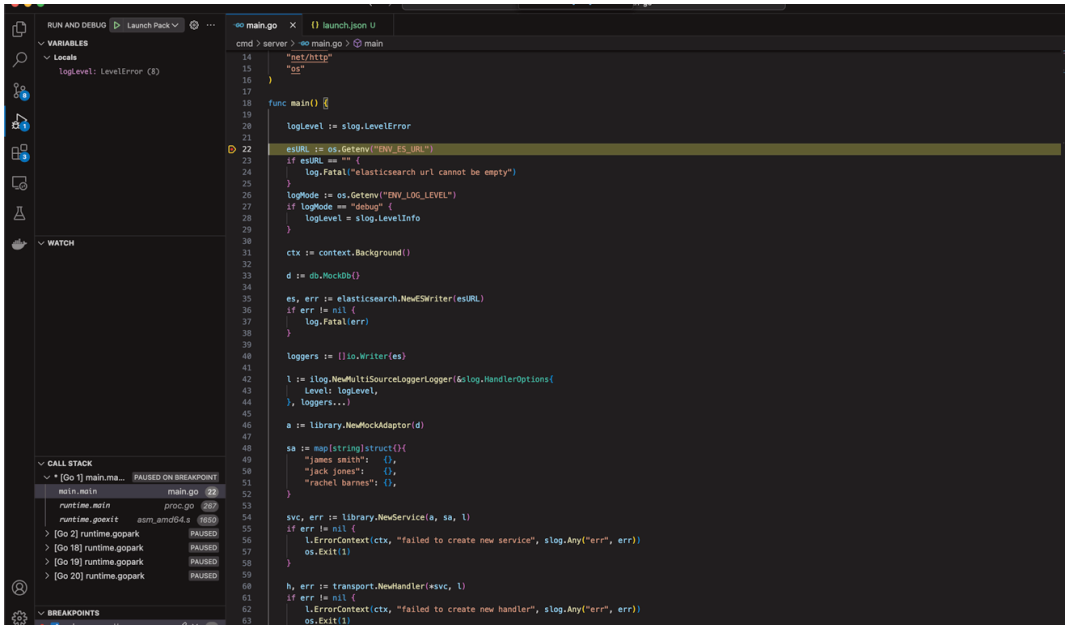
Click "create a launch.json file". Here is a basic one which should help you get started:

```
{
    // Use IntelliSense to learn about possi-
ble attributes.
    // Hover to view descriptions of exist-
ing attributes.
    // For more information, visit: https://go.
microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch Package",
            "type": "go",
            "request": "launch",
            "mode": "auto",
            "program": "${fileDirname}"
        }
    ]
}
```

Once you have copied this, you should have the option to click "launch pack" in the Run and Debug menu:



If you click this, you should see your application begin running, and then pause at your breakpoint as follows:

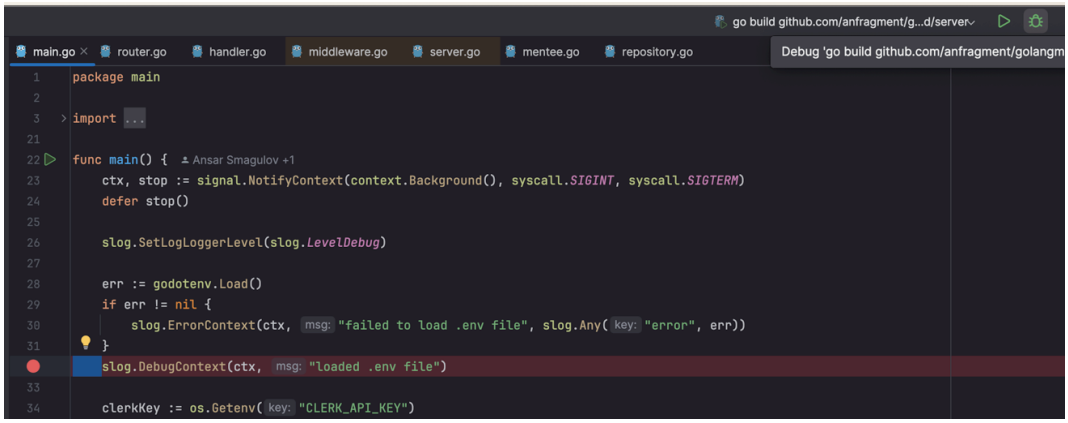


You can see it has highlighted where it was paused and we have some

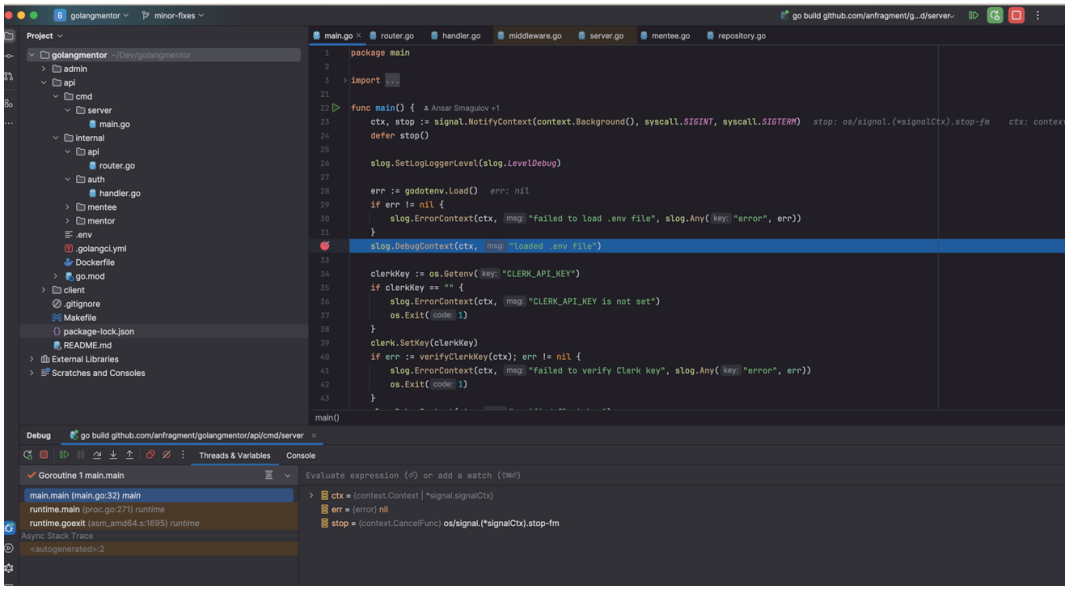
new panels on the left of my screen. One is showing me variables that I have defined and I can take a look at their values.

On the bottom left it shows me the callstack. We can use this to switch between various goroutines, and we will do that a little later in this chapter.

In Goland, you don't need to do any setup here. Simply hit the bug icon in the top right:



Once pressed, your program should hit the breakpoint and show similar highlighting and panels as VScode.



DEBUGGING PANIC TRACES

Let's consider the following Go Program:

```
package main

import "fmt"

func main() {
    numbers := []int{1, 2, 3, 4}
    fmt.Println(numbers[4])
}
```

When we run this code, a panic occurs because our index is out of bounds. Go prints out a trace showing the sequence of function calls that led to the crash.

This example is simple, but you can always read the trace from bottom to top to follow the path of execution. The last function call before the panic shows where the problem originated. It will look something like this:

```
panic: runtime error: index out of range [4] with
length 4

goroutine 1 [running]:
main.main()
/home/user/project/main.go:7 +0x165
```

Although very short, In this trace, we can learn a lot about what went wrong. Let's go through it line by line.

- panic: runtime error: index out of range [4] with length 4:
  This is the panic message. It indicates that the panic was

caused by an attempt to access an index (4) that is outside the bounds of a collection. Likely we had something like a slice with 4 elements in it, but tried to access something at position 5.

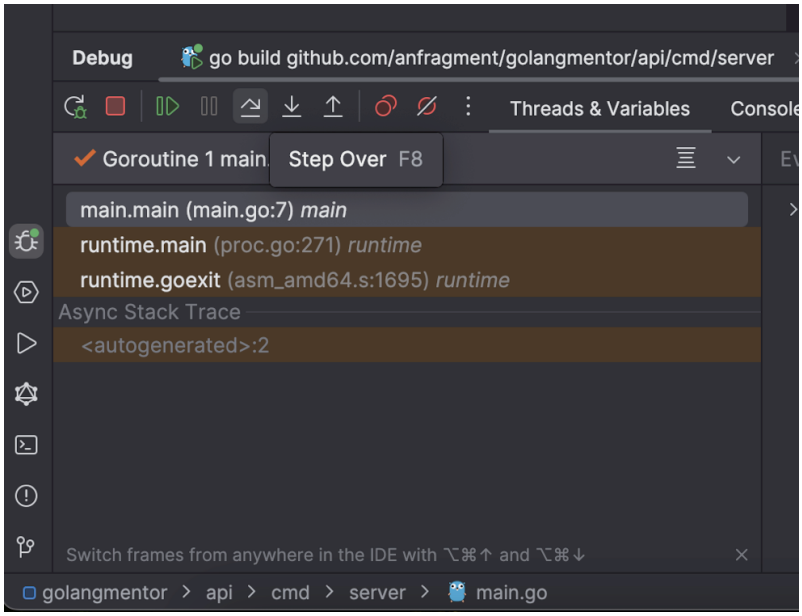
- goroutine 1 [running]:: This shows that the panic occurred in goroutine 1, which is the main goroutine of the program. The [running] status indicates that this goroutine was actively executing at the time of the panic.
- main.main(): This indicates that the panic occurred in the main function of the main package.
- /home/user/project/main.go:7 +0x165: This provides the file name (main.go), line number (7), and an offset (+0x165) where the panic occurred. The offset is the distance in the compiled binary from the start of the function to where the panic happened.

In our IDE, we can also click on the main.go:10 line and it will take us to the exact place the panic occurred in our code. This means we can set a breakpoint before the panic occurs. I set one on line 6:

If we run it in debug mode, we'll then see execution pause before the panic.

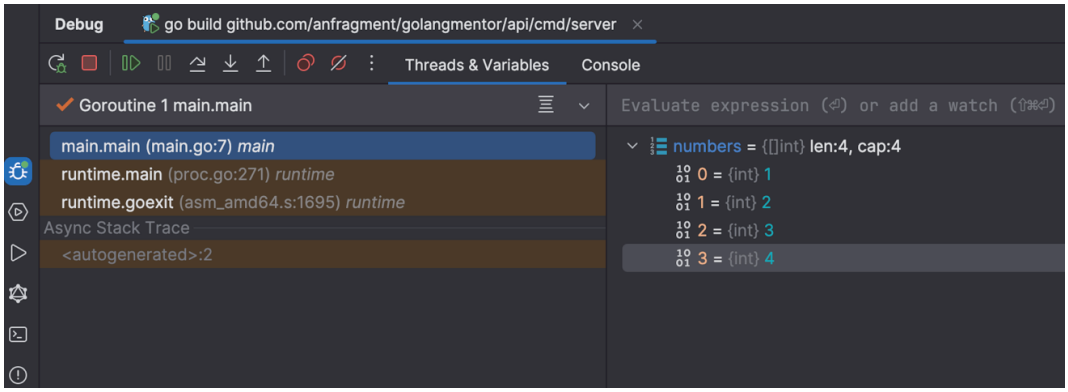
STEPPING OVER

Once execution pauses, we can use the Step Over button to continue execution. In Goland that button looks like this (the slightly bent arrow you can see me hovering over in the image below):



And in VSCode like this (the arched arrow with a dot underneath it).

Once we step over, we'll be able to see our slice of numbers show up in the Evaluation console.



In our simple example, perhaps this doesn't seem that useful. However, imagine a situation where this slice was created dynamically after lots of business logic had been applied. Being able to pause and inspect like this would be highly welcome.

At this point if you step over again, the program will still panic, so hopefully the ability to inspect values has helped you figure out your issue!

STEPPING INTO

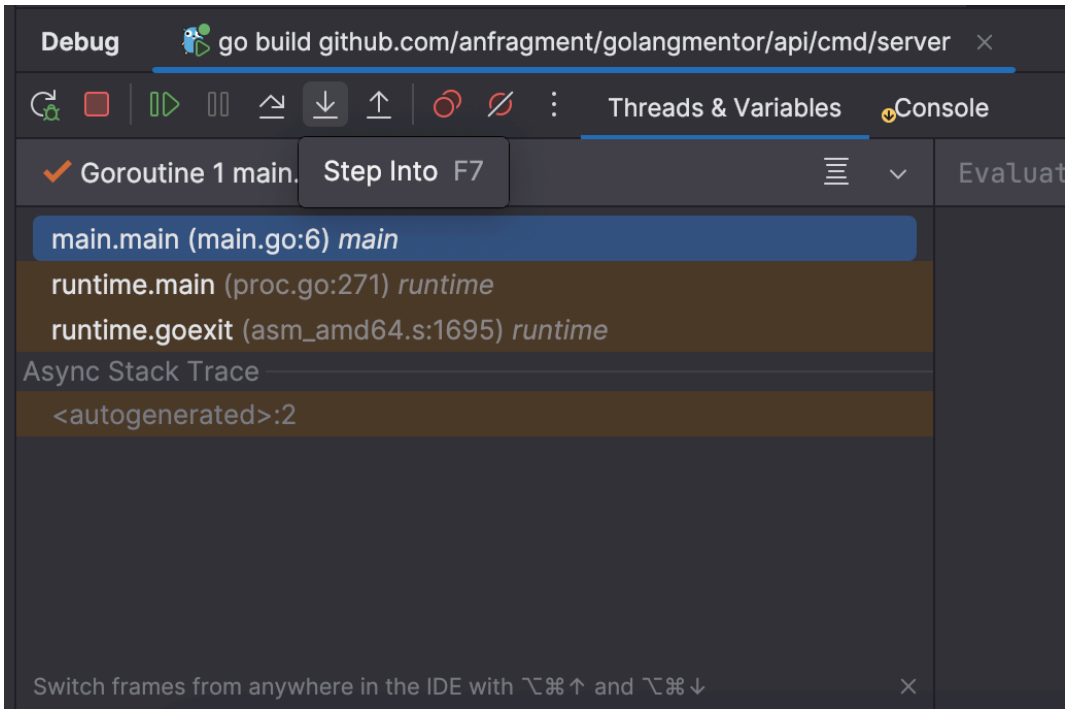
Let's consider a slightly different Go program:

```
func add(a int, b int) int {
    return a + b
}

func main() {
    x := 3
    y := 4
    sum := add(x, y)
    fmt.Printf(
        "The sum of %d and %d is %d\n",
        x, y, sum)
}
```

Let's say we suspect their is a bug with our **`add`** function and set a breakpoint on the line **`y:=4`**. If we keep pressing the step over

button, we will simply skip over the add's function logic and go straight to the **`fmt.Printf`**. Instead, what we want to do is use the "Step Into" button when we reach the line ***`sum := add(x, y).`***

In Goland, that button looks like this (arrow pointing directly down):



And VSCode it looks like this (the arrow pointing down with a dot underneath).

When you press this, you'll jump into the add function and you can step through the logic line by line.

CONDITIONAL BREAKPOINTS

Let's adapt the example above slightly:

```
func add(a int, b int) int {
    return a + b
}

func main() {
    for i := 0; i < 5; i++ {
        x := rand.Intn(10) + 1
        y := rand.Intn(10) + 1
        sum := add(x, y)
    }
}
```

The code is similar, but now we add a for loop and an element of randomness to the sums being calculated. We hear reports that for some reason, we see issues with our code when the value of x is 3. We could modify our program so that x = 3 to debug it, but that risks accidentally "fixing" the bug and the issue not being truly recreated.

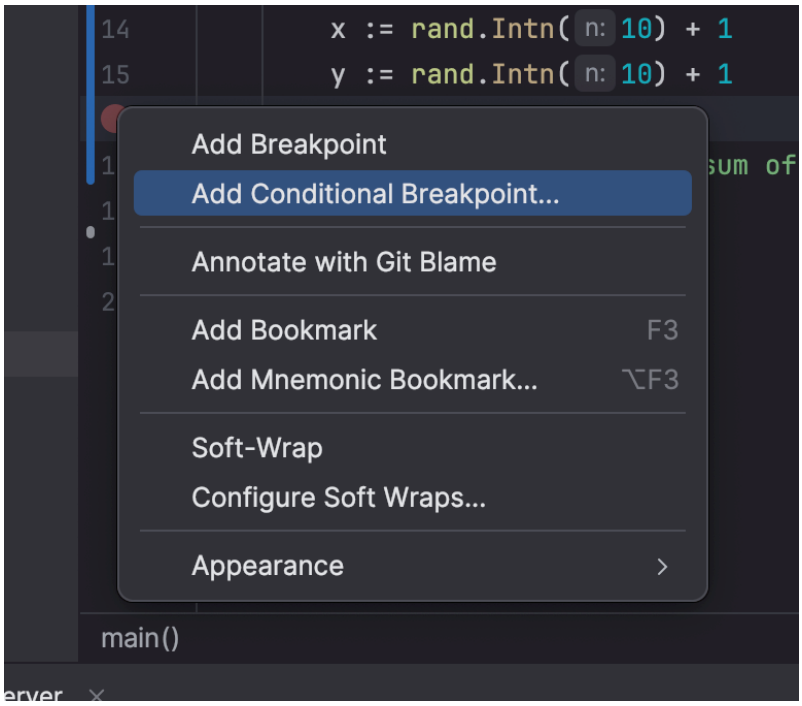
Instead, we can set a conditional breakpoint.

A conditional breakpoint pauses code execution when a condition you specify is true. For example, you may want to pause execution only when:

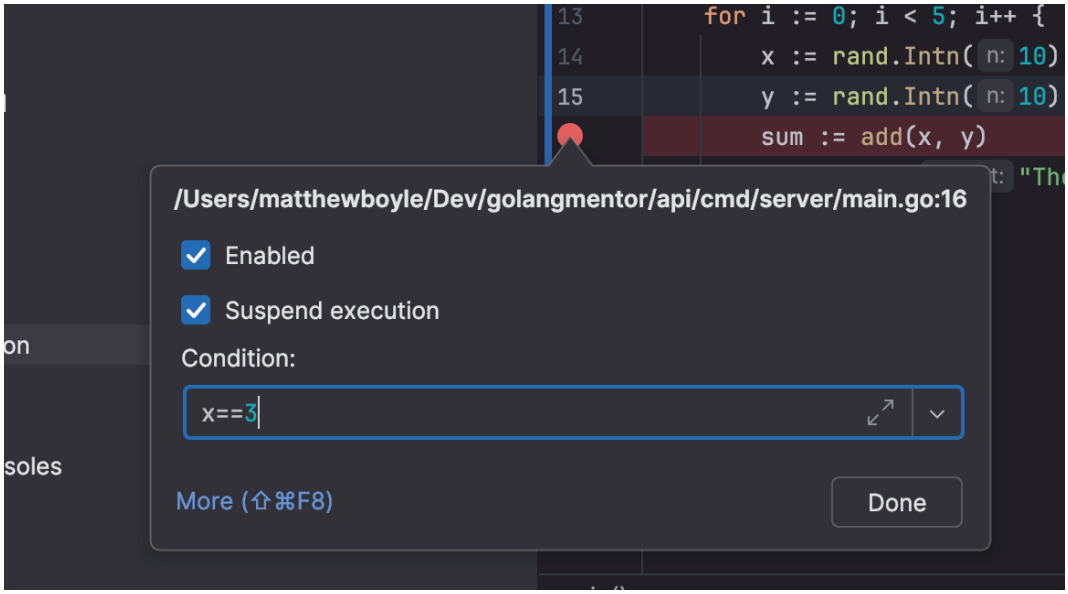
- A variable equals a certain value.
- A function is called with certain parameters.
- An object has a particular property.

The condition is checked each time the breakpoint location is reached. If false, the code continues normally. When true, it pauses execution like a regular breakpoint.

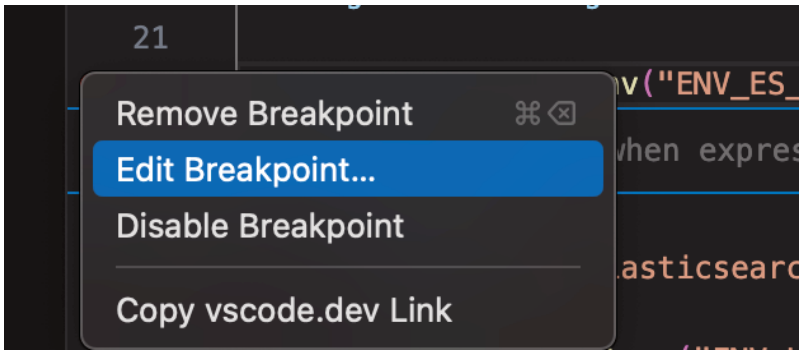
In Goland, you can right-click in the gutter and select add conditional breakpoint:



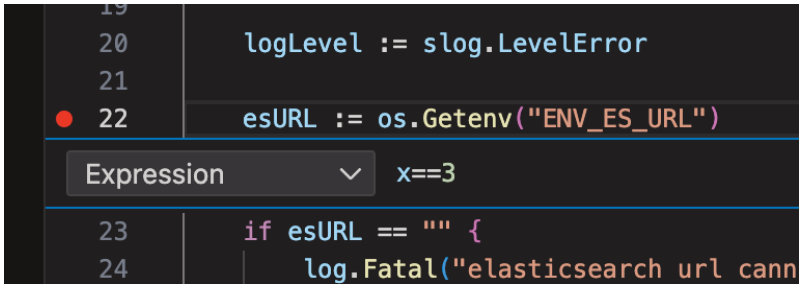
Once selected, you'll see a menu like this where you can enter a condition. Conditions should be boolean, such as **x==3**.



In VSCode, the approach is very similar. If you right click on an existing breakpoint, and select "Edit Breakpoint"



You can then enter a conditional expression such as **`x==3`**:



When you debug your program now, it will only pause execution on this line if this condition is true!

DEBUGGING GOROUTINES

Goroutines are one of the most powerful features of the Go programming language, but they can also be challenging to debug due to their concurrent nature. Even with the same inputs, programs that use the ***go*** keyword may not yield the same outputs every time they are run; the order of execution cannot be guaranteed, making the debugging process more complex.

Let's start with a simple example to illustrate this unpredictable behavior:

```
func printNumbers(prefix string) {
    for i := 0; i < 5; i++ {
        fmt.Println(prefix)
        time.Sleep(time.Millisecond * 10)
    }
}

func main() {
    go printNumbers("A")
    go printNumbers("B")

    // Wait enough time for goroutines to finish
    time.Sleep(time.Second)
    fmt.Println("Main function completed")
}
```

Running this program multiple times may yield different outputs, such as “A B” or “B A”. This inconsistency can make debugging particularly challenging.

While printing to the console is a common debugging technique for concurrent programs, it's often not enough. We need more advanced strategies to identify and inspect specific Goroutines to understand what's happening.

To better understand the challenges of debugging concurrent programs, let's consider a more complex example that follows the worker queue pattern:

```
type Task struct {
    ID      int
    Content string
}

func worker(
```

```
    id int,
    tasks <-chan Task,
    wg *sync.WaitGroup
) {
    for task := range tasks {
        fmt.Printf(
            "Worker %d started task %d\n",
            id,
            task.ID
        )
        processTime := time.Dura-
tion(rand.Intn(5)) * time.Second

        // Simulating task processing time
        time.Sleep(processTime)

        fmt.Printf(
            "Worker %d complet-
ed task %d in %v\n",
            id,
            task.ID,
            processTime
        )
        wg.Done()
    }
}

func main() {
    var wg sync.WaitGroup
    numWorkers := 5
    numTasks := 10

    tasks := make(chan Task, numTasks)
```

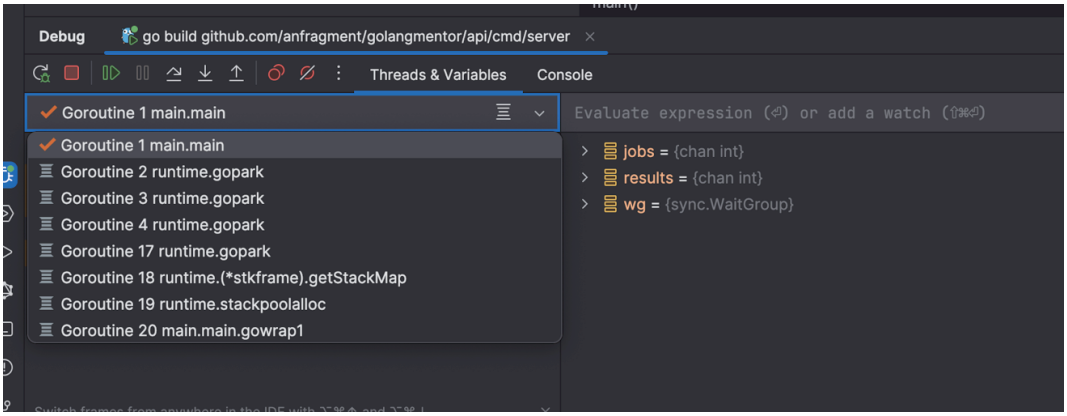
```
    // Start workers
    for i := 1; i <= numWorkers; i++ {
            go worker(i, tasks, &wg)
    }

    // Add tasks to the queue
    for j := 1; j <= numTasks; j++ {
        wg.Add(1)
        tasks <- Task{ID: j, Con-
tent: fmt.Sprintf("Task content %d", j)}
    }

    wg.Wait()
    close(tasks)
    fmt.Println("All tasks completed")
}
```

In this Go program, we:

- Create a slice of tasks to be processed.
- Spawn multiple worker goroutines to handle the tasks.
- Add tasks to a queue for the workers to process.
- Wait for all workers to complete their tasks.
- Print a message indicating that all tasks are completed.

Running this program will output the tasks being processed by each worker, showcasing how different workers may handle more or fewer tasks than others.

Let's say you want to inspect what's happening in a particular goroutine, such as worker three, because you believe it's causing an issue. Setting a breakpoint on **`wg.Wait()`** doesn't necessarily help. Below is a screenshot from Goland once we pause execution. Here we can see a menu for the first time; the goroutine drop down menu.



There are lots of goroutines running, but it's impossible to identify which one is worker 3. Unfortunately it is not goroutine 3. You can read a little bit more about why they don't have predictable names and IDs here[4].

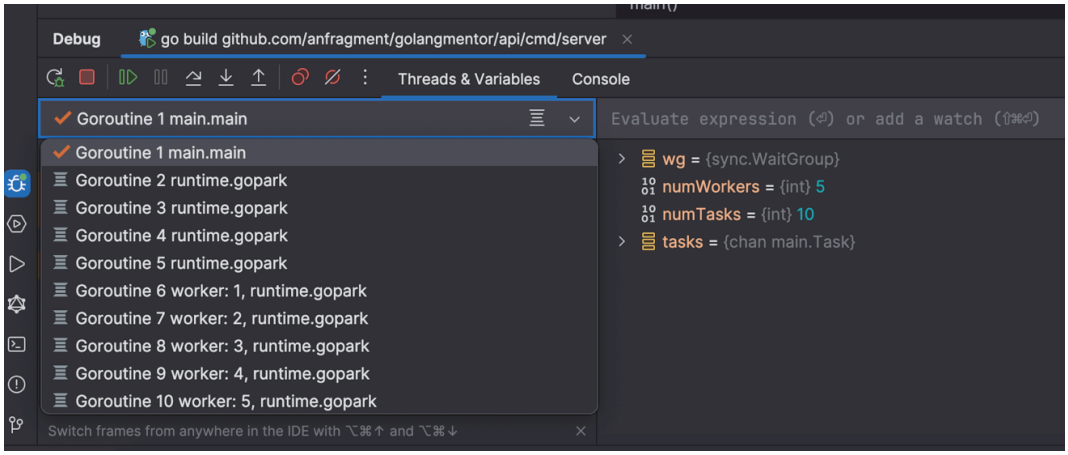
To make our life easier here, we can attach some metadata to the workers here using the **`runtime/pprof`** package. Once imported, we can modify our main function as follows:

```
func main() {
    var wg sync.WaitGroup
    numWorkers := 5
    numTasks := 10

    tasks := make(chan Task, numTasks)

    // Start workers
    for i := 1; i <= numWorkers; i++ {
        labels := pprof.Labels("worker", strcon-
v.Itoa(i))
        pprof.Do(
            context.Background(),
            labels,
            func(_ context.Context) {
                go worker(i, tasks, &wg)
```

```
            }
        )
    }

    // Add tasks to the queue
    for j := 1; j <= numTasks; j++ {
        wg.Add(1)
        tasks <- Task{
            ID: j,
            Content: fmt.Sprintf("Task con-
tent %d", j)
        }
    }

    wg.Wait()
    close(tasks)
    fmt.Println("All tasks completed")
}
```

The key change here is to add labels to our workers using **pprof.Labels**. By doing this, you'll be able to easily identify and locate the specific goroutine you're interested in debugging.

Now, when you debug your program, you'll notice that your goroutines are neatly annotated with labels, making it much easier to find and inspect the one you're looking for.



Being able to find my third worker is now really easy.

Performance Considerations of Labeling

Whilst the labeling trick is incredibly useful, it's essential to be mindful of the potential performance overhead it may introduce. Adding **pprof.Do** could make your program slower, so it's crucial to profile your program carefully if you plan to use this technique. Also, consider removing the labels before releasing to production, as they may clutter your codebase.

TESTS AS AN ENTRYPOINT TO DEBUGGING

In the world of software development, testing is a crucial aspect of ensuring code quality and functionality. However, it's easy to overlook the fact that tests themselves are code too, and they can be just as prone to errors and bugs as application logic.

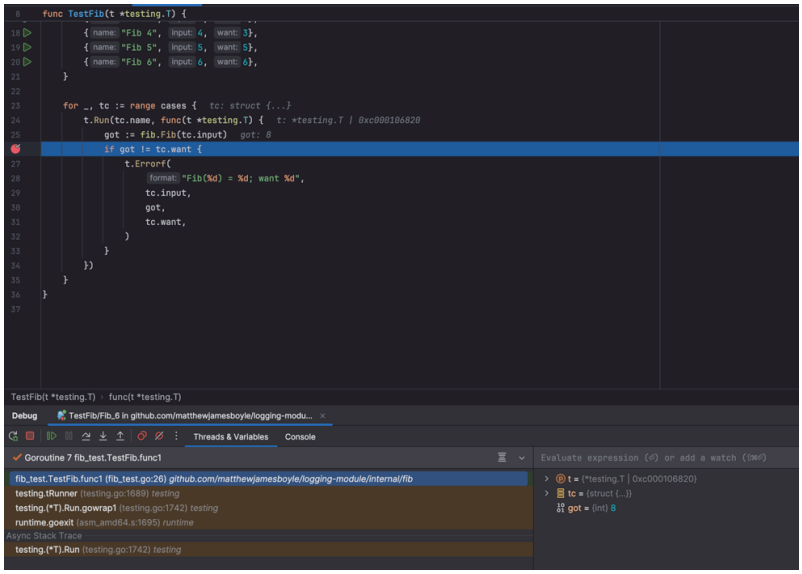
Here's a simple table test in Go with an error in it:

```
func TestFib(t *testing.T) {
    cases := []struct {
        name  string
        input int
        want  int
    }{
        {"Fib 0", 0, 0},
        {"Fib 1", 1, 1},
        {"Fib 2", 2, 1},
        {"Fib 3", 3, 2},
        {"Fib 4", 4, 3},
        {"Fib 5", 5, 5},
        {"Fib 6", 6, 6},
    }
```

```
    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            got := fib.Fib(tc.input)
            if got != tc.want {
                t.Errorf(
                    "Fib(%d) = %d; want %d",
                    tc.input,
                    got,
                    tc.want
                )
            }
        })
    }
}
```

Can you spot the error? (Hint: look at fib 6).

When following Test Driven-Development, the test cases are added first. In cases where, for example, the function being tested is a mathematical algorithm, such as the Fibonacci sequence, it's crucial to verify that our approach is correct. Sometimes, we may have made a mistake in our understanding of the algorithm, which means are tests can be wrong. ***Thankfully***, we can set breakpoints and step through tests, just the same as code itself.



In fact, in almost all situations where you use the debugger, I would advise ensuring you add a test to cover whatever it was you were debugging.

I mentioned the phrase Test-Driven Development (TDD) above; let's talk about that a little more. Whilst debugging is an essential skill, it's often a reactive approach to identifying and fixing issues. To truly embrace a proactive mindset, consider writing your tests first. There are some amazing resources on TDD with Go online, my favorite being Learn Go with Tests[5] which is completely free.

Whilst we are discussing it though, I want to mention some of the benefits as there really does seem to be a correlation between well tested code and lack of defects (and therefore no need to debug!).[6]

Certainty

With a suite of tests, you can confidently make changes to your codebase, knowing that any regressions or unintended consequences will be detected before they reach production. Ideally locally, but if not in your CI pipeline.

Collaboration

Well-written tests serve as documentation for your code, enabling easier collaboration amongst team members and ensuring a shared understanding of the expected behavior. This is especially valuable if you work remotely.

Debugging Entrypoint

As mentioned, tests can act as entry points for debugging, providing a means to observe and analyze your code's execution.

By using the debugger in conjunction with writing tests, you can gain a deeper understanding of how various components interact and how data flows through your application. This knowledge not only assists you in debugging but also empowers you to write more robust and maintainable code.

YOU TRY - EXERCISE

Debugging is best learnt by doing it. Here's an exercise for you to try.

In the repo below, you'll be given the code for the HTTP server with some issues. Your task is to:

- Set up the debugging environment and start the server in debug mode.
- Make requests to the server using a tool like Postman or cURL.
- Set appropriate breakpoints and step through the code to identify and fix the issues.
- Ensure that the server is functioning correctly by making requests and verifying the responses.

You can get the starter code for this exercise from here: https://github.com/MatthewJamesBoyle/ultimate-debugging-course-debug-

module.[7] The code to run is called "exercise". Try and solve all the challenges just with the debugger.

Here's the brief:

Your product manager has created a ticket with the following feedback from your customer.

"I have to be honest, the new TODO app sucks. It seems to crash all the time. Also when I add a new to do, the IDs don't seem to be working quite right. When I go and check my TODOs, it doesn't work at all and on the off chance it does, sometimes I get back an empty response. Can you fix this please?"

After an initial investigation, it seems there are four major issues with the exercise application. Can you fix them all?

Good luck! If you need help or want to validate your answer, there is a video of me walking through the solution here[8]

DEBUGGING IN PRODUCTION

So far we have looked at techniques which will help us debug our application with a focus on local development. Although using the debugger is possible on production and logs are incredibly useful and important, they should form only part of your production debugging strategy.

In this section we are going to look at other tools and techniques you can use to debug your Go application as you start to deploy it into the wild and let real users interact with it.

METRICS

In this chapter, we'll talk about metrics, why they matter, and tips on what to measure.

We'll learn how we can add metrics to our Go applications and I'll share some tips on how you can create dashboards for you to see and monitor them. If your company doesn't use metrics yet, you'll know how to start using them by the end of this chapter.

We'll finish the chapter by having an exercise for you to learn more and get hands-on experience.

I hope you enjoy it and learn a lot!

WHAT ARE METRICS?

Metrics are measurements that give us insight into how our software is performing. They help us understand if our applications are running smoothly or if any issues need attention. Here's a more concrete example.

Imagine you're running a business, and customers can't make successful payments. Every failed payment means lost revenue. By tracking metrics like login failures, successful payments, payment failures, and response times, we can identify issues quickly and understand their impact on our business.

Metrics are not a Go specific thing. What you learn in this chapter can be applied to any programming language.

CATEGORIES OF METRICS

There are a few different categories of metrics we should consider and monitor.

Application Metrics

These metrics help us understand how our application is running. Some important application metrics include:

- **CPU Usage:** High CPU usage can lead to slowdowns or crashes.
- **Memory Usage:** High memory usage can cause performance issues or crashes, and may indicate a memory leak.
- **Goroutines(for Go applications):** Tracking Goroutines helps

us spot leaks, deadlocks, and understand the concurrent nature of our application.

Business Metrics

While application metrics are important, we write software to solve business problems. That's why we need to track business metrics too. These will vary, but some things to track might be:

- Login failures.
- Successful payments.
- Response times.

These metrics help us understand the impact our application is having on our users and business value.

Infrastructure Metrics

Depending on your infrastructure setup, you may want to track additional metrics. For example, if you're running on Kubernetes, you might want to monitor:

- Number of running pods (zero pods could indicate a problem).
- CPU throttling (if you've hit your CPU limits, Kubernetes may throttle your application).
- Disk I/O (important if your application reads and writes from disk frequently).
- Job completions (for cron jobs or batch processes).

This seems an awful lot of things to think about! Whilst this is true, thankfully, at least in Go, measuring a lot of these things is very easy due to the Prometheus[1] library. Prometheus is an open-source, real-time monitoring and alerting toolkit that is well-suited for cloud-based environments. Prometheus stores metrics as time-series data,

with each data point including a timestamp and optional labels. This allows for powerful querying using PromQL, a query language that lets you select and aggregate data based on labels.

We'll learn how to use PromQL a little bit later to build Grafana dashboards and visualize our metrics.

Now we understand what metrics are and some of the things we might measure, let's look at the different type of Prometheus metrics we can use in more detail.

PROMETHEUS METRIC TYPES

Prometheus supports several types of metrics, each with its own purpose. Let's explore the basic ones:

Counters

These metrics represent a single number that can only increase over time. An example would be tracking successful transactions. Each time a transaction is successful, the counter goes up.

Gauges

Unlike counters, gauges represent a single number that can go up or down. For instance, you could use a gauge to monitor the amount of requests currently being received or your blood sugar level, which fluctuates based on various factors[2].

Histograms

These metrics are handy for tracking observations like request durations or response sizes. They count the observations in different buckets, providing a summary of all the observed values. For example, you could use a histogram to monitor how quickly your application responds to requests.

Summaries

Similar to histograms, summaries calculate configurable quantiles (like percentiles) over a sliding window of observations. While histograms are often sufficient, summaries can be useful in certain situations.

Metrics can be even more informative when you add labels to them. Labels act like dimensions, allowing you to split your metrics based on different criteria. For instance, you could label transaction success metrics by payment method (Visa, Amex, MasterCard, etc.).

Enough theory - let's see some Go code.

ADDING METRICS TO A GO APPLICATION

Defining Metrics

After adding the Go Prometheus library[3] as a dependency, we need to define our metrics. Here's an example of how to define a counter for tracking successful transactions.

```
var transactionSuccessCounter = prometheus.New-
Counter(prometheus.CounterOpts{
    Name: "transaction_success_total",
    Help: "The total number of successful trans-
actions.",
})
```

Notice how we give our metric a name and a help description. These details make it easier to understand what the metric represents.

Registering Metrics

After defining our metrics, we need to register them with Prometheus. This can be done using the prometheus.Register function:

```
err := prometheus.Register(transactionSuccess-
Counter)
if err != nil {
    // Handle registration error
}
```

Alternatively, you can use **prometheus.MustRegister()**, which will panic if any of the metrics are nil. I prefer to handle errors explicitly, so I rarely use this.

Using Metrics In Your Code

Once our metrics are registered, we can start using them in our application's code. Let's say we have a function that handles successful transactions, such as:

```
func handleSuccessfulTransaction() {
    // Logic for handling successful transaction
    transactionSuccessCounter.Inc()
}
```

Here, we increment the **transactionSuccessCounter** every time a transaction is successful.

As mentioned previously, there are other metric types beyond counters.

Gauges

Gauges are useful for tracking values that can fluctuate, like blood sugar levels. Here's an example of how to define and use a gauge:

```
var bloodSugarGauge = prometheus.New-
Gauge(prometheus.GaugeOpts{
    Name: "blood_sugar_level",
    Help: "The current blood sugar level.",
})
```

```
func updateBloodSugar(level float64) {
    bloodSugarGauge.Set(level)
}
```

In this example, we define a gauge called **bloodSugarGauge** and use the Set method to update its value based on the current blood sugar level. This could be very useful for an application tracking health vitals.[4]

Histograms

Histograms are great for tracking observations like request durations or response sizes. Here's an example of how to define and use a histogram:

```
var requestDurationHistogram = prometheus.NewHis-
togram(prometheus.HistogramOpts{
    Name: "request_duration_seconds",
    Help: "The duration of HTTP re-
quests in seconds.",
    Buck-
ets: []float64{0.01, 0.05, 0.1, 0.5, 1.0, 3.0},
})

func handleRequest(w http.Response-
Writer, r *http.Request) {
    start := time.Now()

    // Handle request logic

    duration := time.Since(start).Seconds()
    requestDurationHistogram.Observe(duration)
}
```

In this example, we define a histogram called **requestDuration-Histogram** with buckets for different request durations. In our

request handler, we measure the duration of the request and observe it using the Observe method.

Summaries

In my experience, summaries are rarely used and histograms are your best bet for most scenarios. If you want to dig into where summaries might be useful, this[5] StackOverflow answer is very good. If you end up with the rare use case of wanting to use one, here is how you can do it.

```
var responseSizeSummary = prometheus.NewSummary(
    prometheus.SummaryOpts{
        Name: "response_size_bytes",
        Help: "The size of HTTP respons-
es in bytes.",
        Objectives: map[float64]float64{
            0.5: 0.05,
            0.9: 0.01,
            0.99: 0.001,
        }, // Adjust objectives as needed
    },
)

func handleResponse(w http.Response-
Writer, r *http.Request) {
    // Handle response logic
    response-
Size := // get the size of the response
    responseSizeSummary.Observe(
        float64(responseSize)
    )
}
```

In this example, we define a summary named **responseSizeSummary** with objectives (quantiles) for different response sizes. In our

response handler, we observe the size of the response using the **`Observe`** method.

EXPOSING METRICS

Now we know how to register and increment metrics, we need to ensure we expose them so that they are available for us to view and make decisions on. The easiest way to do that is to expose them via HTTP. Thankfully, the Prometheus library makes this really easy to do in Go!

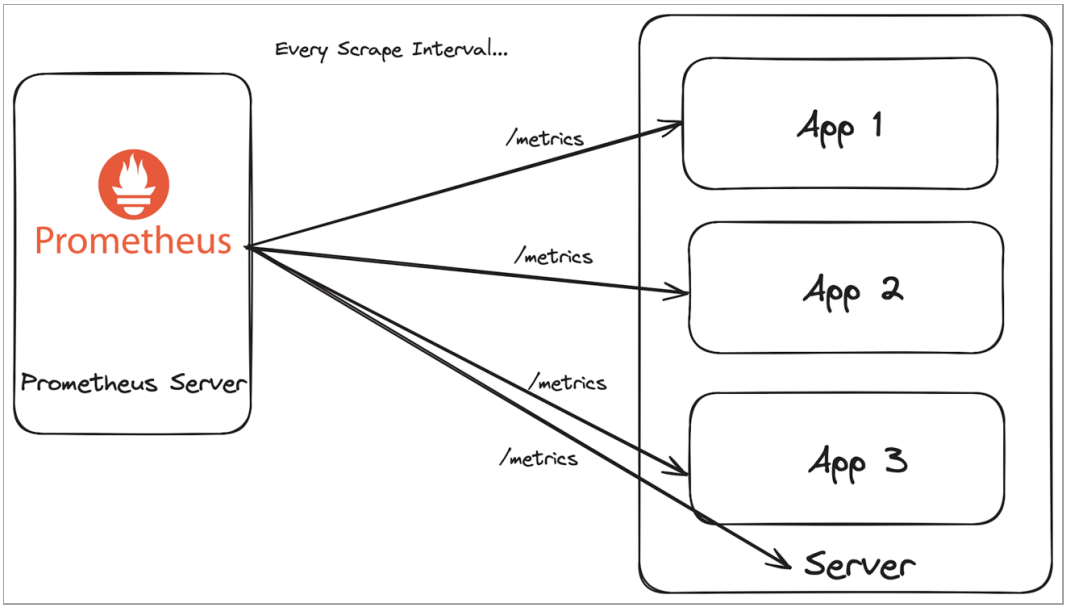
```
import "github.com/prometheus/client_-
golang/prometheus/promhttp"

func main() {
    // Register metrics (as shown earlier)
    http.Handle(
        "/metrics",
        promhttp.Handler()
    )
    log.Fatal(
        http.ListenAndServe(":8080", nil)
    )
}
```

In this example, we use the **promhttp.Handler()** function to create a new HTTP handler that serves our registered metrics at the /metrics endpoint. When Prometheus scrapes this endpoint, it will collect all the metrics we've defined and registered.

What do you mean when Prometheus scrapes this endpoint?

Prometheus collects metrics from targets by scraping metrics HTTP endpoints. This is powerful, because it means once you (or your SRE team) have configured Prometheus, all your applications (and servers)

can be automatically discovered without any other changes. If you're a small team, Prometheus is available as a managed service from AWS[6], GCP[7] or Azure[8]. Some tools such as Grafana or Datadog can also take the output of this information and store it in their cloud (more on this later).



As noted in the diagram above, we can define a scrape interval for Prometheus and that determines how often Prometheus will reach out and scrape our metrics endpoint.

VIEWING METRICS

We have talked about the **`/metrics`** endpoint a lot, so let's actually see one! If you have been following along and have been adding some metrics for a go application, go ahead and run it and browse to **`http://localhost:8080/metrics`** in your browser. You can also clone the repo I prepared here[9].

Once you navigate to **`/metrics`**, you should see something like this (I trimmed it for succinctness).

```
# HELP go_gc_duration_seconds A summa-
```

```
ry of the pause duration of garbage collec-
tion cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.000108247
go_gc_duration_seconds{quan-
tile="0.25"} 0.000123756
go_gc_duration_seconds{quan-
tile="0.5"} 0.000128698
go_gc_duration_seconds{quantile="0.75"} 0.0001423
go_gc_duration_seconds{quantile="1"} 0.000245674
go_gc_duration_seconds_sum 0.001883626
go_gc_duration_seconds_count 12

# HELP go_goroutines Number of gorou-
tines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 23

# HELP go_info Information about the Go envi-
ronment.
# TYPE go_info gauge
go_info{version="go1.22”}

# HELP transaction_success_total The total num-
ber of successful transactions.
# TYPE transaction_success_total counter
transaction_success_total 47
...
```

At the very top, you'll see some general metrics about our program, like how many goroutines are running and which version of the Go programming language we're using. The really cool thing about this is we get all of this information for free, just by calling **promhttp.Handler()**. You'll also notice that **transaction_success_total**, our custom metric appears too.

We can now see some metrics on an endpoint, but the single values it provides are not all that useful. Prometheus stores the data from this endpoint in its database and that means we can do some time-based queries. Let's take a look at how to do that and then discuss how that will help us debug production.

INTRODUCTION TO PROMQL

PromQL is the query language used by Prometheus.

To write effective queries, you must first understand the structure of the data you are querying. Each metric in Prometheus is stored as a time series, labeled with names that describe what the data represents, as we saw previously.

When writing a query, the simplest form is to specify the metric name. This retrieves a series of data points (values with timestamps) for that metric. For example:

```
http_requests_total
```

For more targeted data retrieval, you can use braces **{}** to filter by specific labels.

Here's an example:

```
http_requests_total{service="user-service",
method="GET"}
```

This query will return the total count of GET requests up to the current point in time for a service called user-service.

You can also make queries such as:

```
sum(rate(http_responses_total{status=~"5.."}
[5m]))
```

There's a lot going on here that shows the value of Prometheus, so lets talk through it bit by bit:

```
http_responses_total
```

This is the name of the metric being queried. In this case, http_responses_total likely records the total number of HTTP responses generated by your application.

```
{status=~"5.."}
```

We call the piece inside `{}` the label selector. `status=~"5.."` is a label filter using a regular expression. The =~ operator matches label values that fit the regex pattern provided. Here, "5.." matches any status code that begins with '5', which typically represents server error responses (like 500 Internal Server Error, 502 Bad Gateway, etc.). The result is that this part of the query filters http_responses_total to only consider metrics where the status label starts with a '5', effectively selecting only error responses.

```
[5m]
```

Anything within [] is a range vector. This specifies the time range over which to evaluate the metric. [5m] means "the last 5 minutes." When applied to a metric, this creates a range vector, which includes data points for each instance of the metric within the last 5 minutes.

```
rate()
```

rate() is a function used specifically with counters (metrics that only increase over time, like `http_responses_total`). It calculates the per-second average rate of increase of the metric over the specified time range, in this case, 5 minutes. This is particularly useful for understanding the rate of change of a metric, smoothing out any spikes that might have occurred in a short period. This smoothing is important if you want to use the query to call out an Engineer if it goes out of bounds for a period of time; a common use-case.

sum()

sum() is an aggregation operator that sums up all the values of the resulting vector from the `rate()` function. In this query, it sums up the rates of HTTP error responses from all collected data points (or possibly across different dimensions, like different servers or service instances, if your metric includes those labels but they aren't filtered out in this query).

PromQL can be a little challenging to get started with, but as you can see it is very powerful. Typically when you deploy Prometheus or use a managed service, you will get access to the Prometheus UI which can be used for entering queries such as those discussed above:

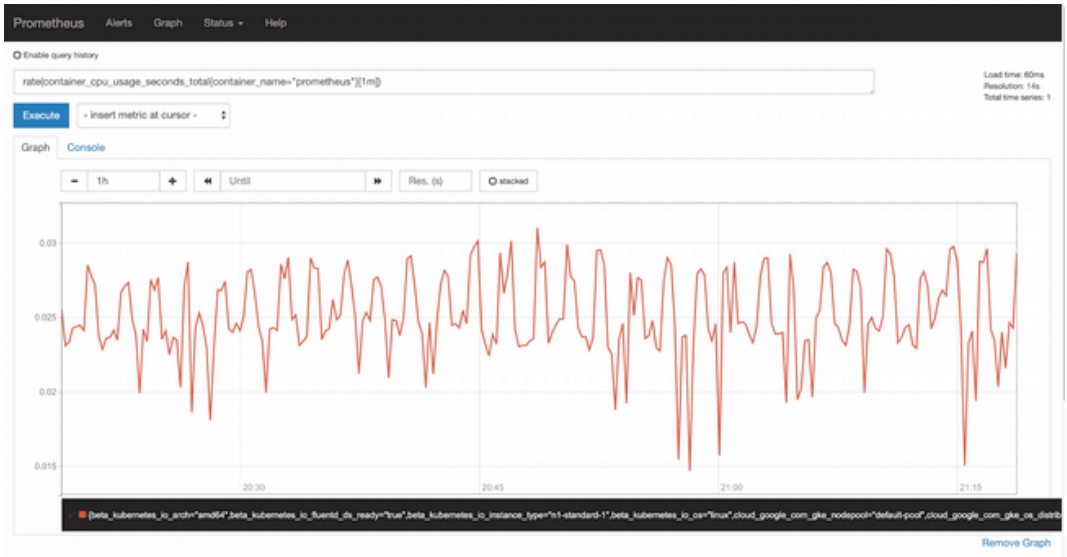


Image credit: opensource.com

PromQL is something I have never truly "learnt" beyond the basics, and I use this handy reference sheet[10] regularly; it's very useful.

USING METRICS TO HELP YOU DEBUG PRODUCTION

We now know everything we need to start measuring our Go application's behavior. We can now get insight into business metrics as well as application metrics without having to use the debugger, logs, or running it locally.

Here are some application queries that will help you get started with tracking your Go application.

Goroutine Count

Tracks the current number of goroutines, useful for spotting spikes in real-time.

```
go_goroutines
```

Heap Usage

Monitor the amount of heap memory currently in use.

```
go_memstats_heap_alloc_bytes
```

Garbage Collection Pause Duration

Track the 90th percentile of GC pause durations over the last 5 minutes. This can be useful for spotting performance bottlenecks

```
histogram_quantile(0.9, rate(go_gc_duration_sec-
onds_bucket[5m]))
```

CPU Usage

Monitor average CPU usage per second over the last 5 minutes .

```
rate(process_cpu_seconds_total[5m])
```

HTTP Request Rate

Calculate the rate of HTTP requests per method over the last 5 minutes.

```
sum(rate(http_requests_total[5m])) by (method)
```

HTTP Error Rates

Calculate the proportion of HTTP responses with 5xx status codes over the total number of responses in the last 5 minutes.

```
sum(rate(http_responses_total{status=~"5.."}
[5m])) / sum(rate(http_responses_total[5m]))
```

HTTP Request Latency

Measure the 95th percentile of HTTP request latency over the last 5 minutes.

```
histogram_quantile(0.95, rate(http_request_dura-
tion_seconds_bucket[5m]))
```

Thread Count

Monitors the number of active threads in use by the Go process.

```
process_threads
```

Open File Descriptors

Track the number of open file descriptors, which should not approach the limit set by your system.

```
process_open_fds
```

It might not be immediately clear why measuring all of these things is helpful. The important thing is to look for patterns over time. When your system gets slow, or just before it crashes, what was it reporting? This gives you a jump off point to begin your investigation.

Ideally, we do not want to have to open Prometheus and run queries constantly to make sure our application is performing ok. I have found Grafana is great for building dashboards that can be setup to live refresh, and give you a holistic view of how your application's performing in one place.

Example Grafana dashboard, courtesy of grafana.com

I tried to write a section on creating Grafana dashboard but I didn't think it worked well in book form - I do cover it in the course and you will get an opportunity to do this in the exercise too.

ALERTING

Grafana[11] and Prometheus[12] not only provide visualization capabilities but also offer alerting and automation features. You can define alert rules based on specific metric conditions, such as transaction success rates dropping below a certain threshold or HTTP request durations exceeding a predefined limit.

These alerts can be configured to trigger notifications through various channels, including email, Slack, PagerDuty, or custom webhooks. This allows you to be proactively notified of potential issues, enabling faster response times and reducing the risk of prolonged outages or performance degradation.

Additionally, you can integrate these alerting systems with automated remediation or self-healing processes, further enhancing the resilience and reliability of your application.

EXERCISE

Using this repo[13], the goal is to build a dashboard in Grafana to track the number of goroutines currently in use in the application. The README gives more instructions on how to get started.

Your graph should look pretty boring. Your challenge is to introduce some code to one of the endpoints that causes the amount of goroutines being used to increase over time. Effectively, I want you to introduce a bug to the code base. This might seem counterintuitive, but understanding how a bug can exist and what it would look like on your Grafana dashboard can be really helpful when it happens in a less controlled scenario.

Once you're done, you can watch me walk through a solution here[14].

Good luck!

A WARNING ON MEASURING TOO MUCH

With great power comes great responsibility. When you first learn about metrics, it can be tempting to have a metric for everything (the same way that you start by logging everything in the beginning).

Here's a couple of things to be aware of as you start to instrument your application.

System Overload:

Tracking too many metrics in Prometheus can strain your system. It can slow down, be less responsive, and overall be less efficient. This could affect not just monitoring but also the stability of the system you're monitoring.

Higher Costs and Complexity

More metrics mean needing more resources for storage and processing, which costs more. It also makes managing your monitoring system more complicated.

Hard to Find Important Data

With so many metrics, it's tough to spot the really important information. This can slow down how quickly you respond to serious issues in your system.

Maintenance Gets Tougher

The more metrics you have, the harder it is to keep everything up to date and relevant. Too many metrics can make it hard to focus on what's truly important for your system's health.

It’s not all doom and gloom though. Here are some tips to help make sure you do measure the right things.

Focus on Key Metrics

Only track metrics that are really useful for understanding your system.

Regularly Review Your Metrics

Get rid of metrics that aren't helpful anymore.

Use Aggregation When Possible

This can reduce the amount of data you need to store and process.

Keep an Eye on Prometheus Itself

Make sure your monitoring tool isn’t using too many resources.

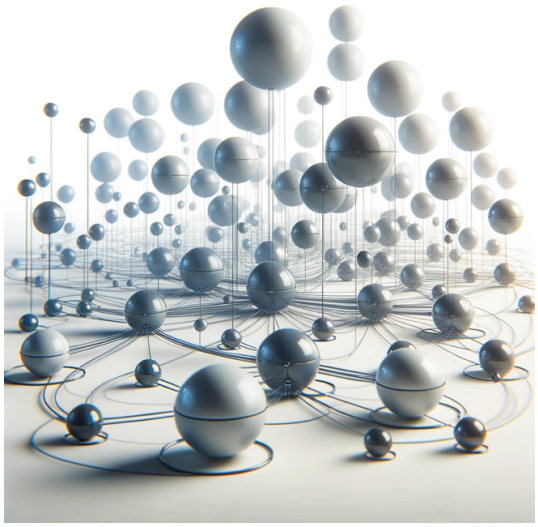
Remember, effective monitoring and debugging is about getting valuable insights, not just collecting lots of data.

You are now a metrics expert! I hope this chapter has given you some

concrete actions you can take away that will give you valuable insight into your production systems.

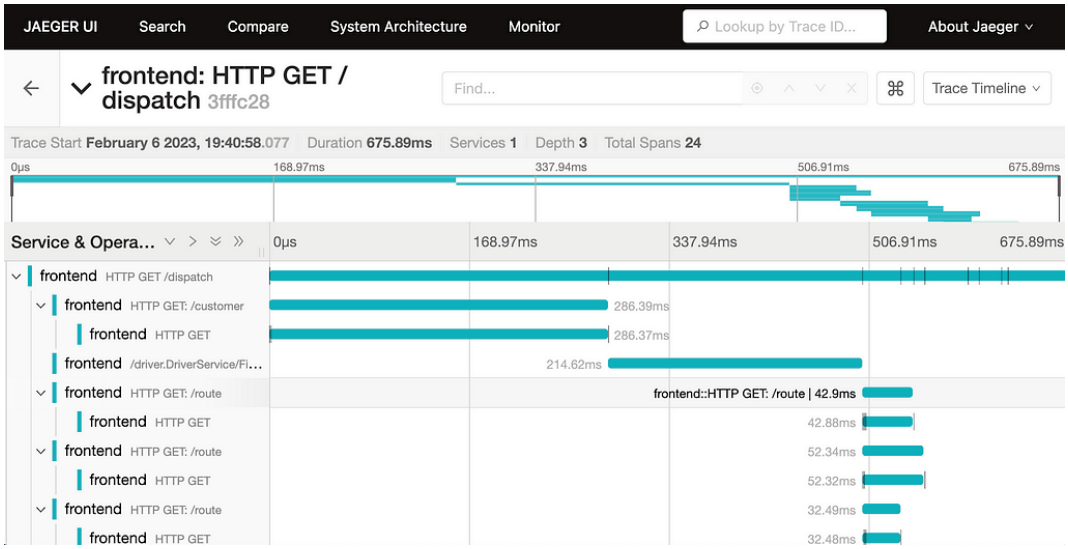
In the next chapter we are going to discuss distributed tracing. I'll see you there!

DISTRIBUTED TRACING



As systems become distributed, they become harder to reason about. This book is not about whether you should build microservices or a monolith - there are plenty of books on that! However, we must acknowledge that at some point in your career you will likely work in a place where the business logic is distributed across multiple systems. When this happens, everything we have learnt up to now is still valuable but we likely want to invest further to ensure we can still debug our systems across system boundaries.

Distributed tracing provides a method to track and visualize requests as they traverse the various components of a distributed architecture. Picture a breadcrumb trail spanning the entire application and its underlying infrastructure, allowing you to trace the journey of a request from start to finish. This visibility is crucial for comprehending system behavior and pinpointing and resolving complex issues.



An example distributed trace; image courtesy of jaegartracing.io

The concept of distributed tracing has its roots in **Google's Dapper project**[1], a large-scale tracing system that inspired numerous open-source initiatives. From the concepts of Dapper, tools like **Zipkin and Jaeger** emerged, quickly becoming prominent players in the tracing world. The advent of standards like **OpenTelemetry** further simplified the implementation of tracing across various technologies, including Go, which boasts excellent support for distributed tracing.

OPEN TELEMETRY

OpenTelemetry(often referred to as OTel) is an open-source project designed to make observability simple. It allows developers to collect, analyze, and export telemetry data such as metrics, logs, and, the focus

of this chapter, traces. The goal is to help you monitor your software and debug issues more effectively.

Generally, Open Telemetry is split into two key components.

Code Emitters

These are the components within your application that emit traces.

Collectors

These are infrastructure components deployed to collect and process the emitted traces.

In the exercise a little later, we will be using Jaeger as our collector, although other paid options are available, such as:

- Datadog
- Honeycomb.io
- AWS offerings

Enough background. This book is about Go!

ADDING TRACES TO A GO API

I have prepared a repository for this section that you can use to follow along.[2] However, all relevant code snippets will be included.

Firstly, let's get Jaegar started on our machine. One simple way to do this is use this Docker Compose file[3]:

```
version: '3'
services:
  jaeger:
    image: jaegertracing/all-in-one:latest
    ports:
      - "16686:16686" # UI
      - "14268:14268" # Collector
```

```
      - "14250:14250" # gRPC
      - "9411:9411"   # Zipkin
```

This Docker Compose file pulls the **jaegertracing/all-in-one** image, which includes the following components:

• User Interface (UI)

• Collector

• Query

• Agent

Once you've saved the Docker Compose file, simply run ***docker-compose up*** in your terminal, and you'll have a fully functional Open-Telemetry stack running locally. You should be able to access the Jaeger UI by navigating to **http://localhost:16686** in your web browser.

Let's start by considering a simple HTTP server and endpoint. It would look something like this.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", SimpleHandler)
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal(err)
```

```
    }
}

func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    _, _ = w.Write(
        []byte("Hello, World!")
    )
}
```

To add tracing, we need to import a couple of libraries and add some configuration. Let's do that now.

```
package main

import (
    "go.opentelemetry.io/contrib/instrumenta-
tion/net/http/otelhttp"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/jaeger"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/sem-
conv/v1.7.0"
    trace2 "go.opentelemetry.io/otel/trace"
    "log"
    "net/http"
)

var tracer trace2.Tracer

func main() {
    exporter, err := jaeger.New(
        jaeger.WithCollectorEndpoint(),
    )
```

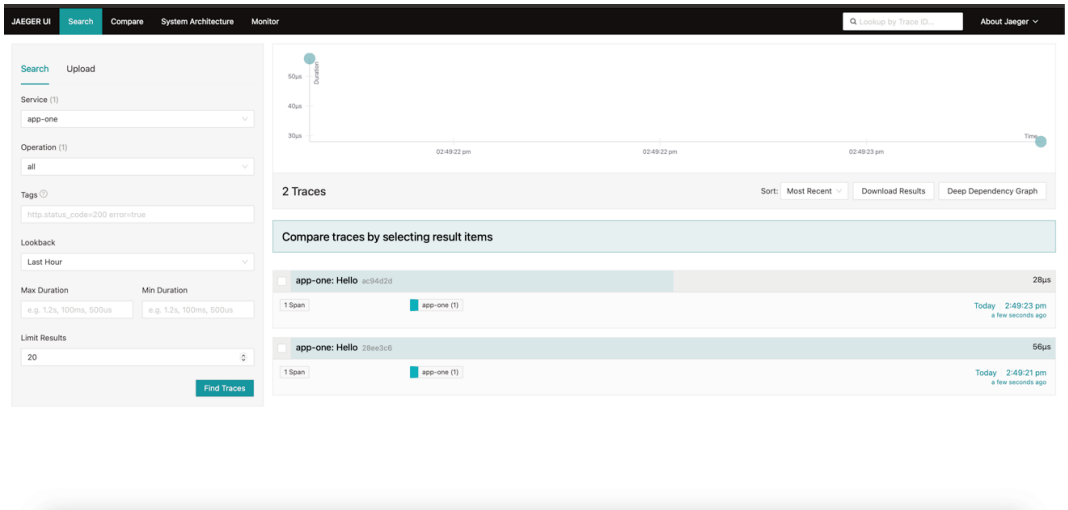
```
    if err != nil {
        log.Fatal(err)
    }

    // Create Trace Provider
    tp := trace.NewTracerProvider(
        trace.WithBatcher(exporter),
        trace.WithResource(resource.NewWithAt-
tributes(
            semconv.SchemaURL,
            semconv.ServiceNameKey.String("app-
one"),
        )),
    )

    otel.SetTracerProvider(tp)
    tracer = tp.Tracer("app-one")

    http.Handle("/",
        otelhttp.NewHandler(
            http.HandlerFunc(SimpleHandler),
            "Hello",
        ),
    )
  err := http.ListenAndServe(":8080", nil)
    If err != nil {
        log.Fatal(err)
    }
}

func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    _, _ = w.Write(
        []byte("Hello, World!"),
    )
```

```
}
```

We've added Jaeger as the tracing backend and created a new HTTP handler with tracing middleware using OpenTelemetry's **`otelhttp.NewHandler`** function. This middleware will trace incoming requests to our endpoint. If you are not familiar with middleware, they act as a layer that processes requests and responses in a web application, allowing for various operations such as logging, authentication, or, in this case, tracing. Middleware functions can be composed to handle different aspects of request processing, providing a modular and reusable approach to managing cross-cutting concerns in your application.

You can run this code and access the "/" endpoint. Jaeger will start collecting traces, and you can visualize them in the Jaeger UI. If everything runs as expected, when you make a request to http://localhost:8080/ in your browser, you should also see a trace appear in the Jaegar UI when you access it. If you used my **`docker-compose`** file, it should be available at **`http://localhost:16686/`**.

You should see something like this:



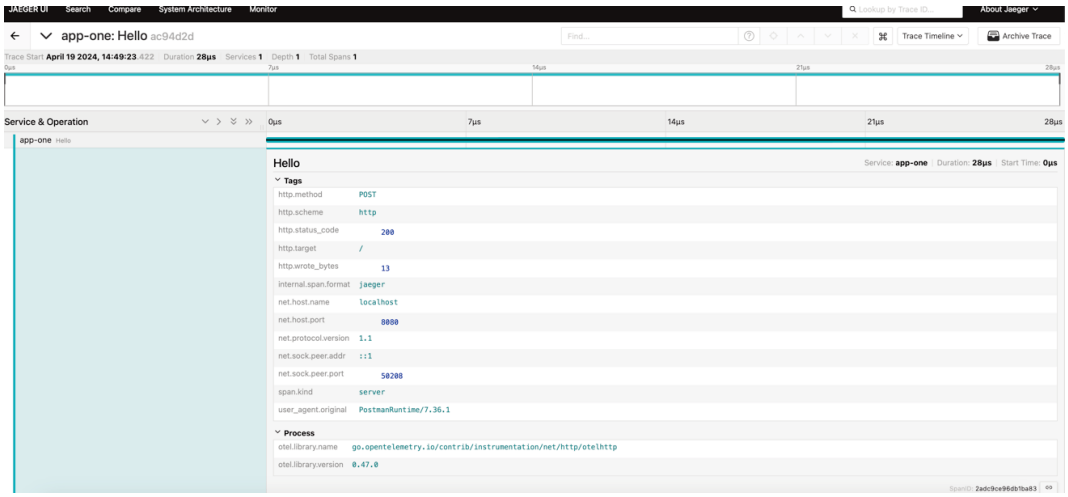
Pretty underwhelming for now, but it proves everything is working!

The eagle-eyed amongst you might have noticed we didn't specify an endpoint to send the traces to. That's because **`jaeger.WithCollectorEndpoint()`** defaults to send traces to **`localhost:14268`**, which is the same as what we defined the collector to be in our docker-compose file. We also defined:

```
var tracer trace2.Tracer
```

But are not using it yet. We'll use it shortly.

Even with this very basic example, you can see some of the promise that distributed tracing can bring. If you click on one of the traces in the UI and expand it, you'll see the following:



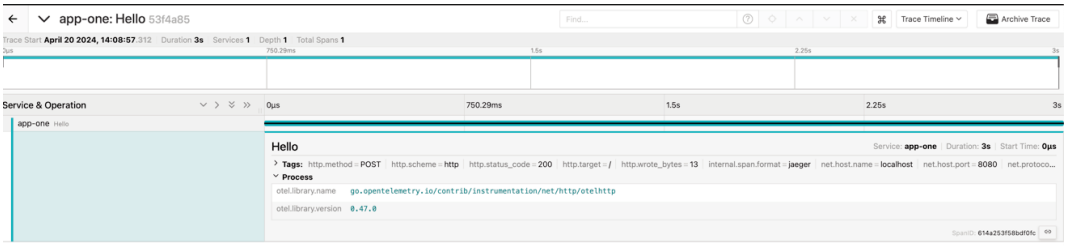
We can see lots of interesting metadata about the request, including the status code received, the name of the span and how long it took our service to respond.

While this simple instrumentation provides valuable insights, the true power of distributed tracing shines when dealing with more complex scenarios. Let's explore how to instrument a more involved endpoint that simulates a database call. Let's add an artificial delay to our **`SimpleHandler`** function:

```
func SimpleHandler(w http.Response-
```

```
Writer, r *http.Request) {
    // imagine this is a slow DB query.
    time.Sleep(time.Second * 3)
    _, _ = w.Write([]byte("Hello, World!"))
}
```

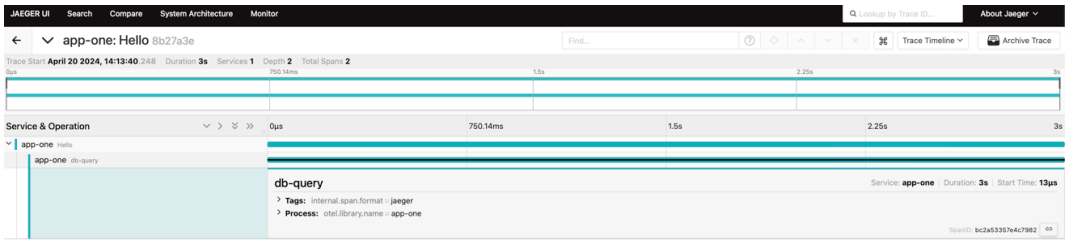
If we make a request again, we can see that the span is now 3 seconds long, but it doesn't really help us debug why it is 3 seconds long.



We can use the tracer we created before to define a custom span as follows:

```
func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    _, span := tracer.Start(
                r.Context(),
                "db-query",
                )

    // imagine this is a slow DB query.
    time.Sleep(time.Second * 3)
    span.End()
    _, _ = w.Write([]byte("Hello, World!"))
}
```

Now if we re-run our application and make another request. You'll see a span that looks like the following:



This is really powerful. We can now see app-one has a span within it called db-query that is causing almost all of the processing time. If we put an artificial delay of a second before the db-query span as follows:

```
func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    time.Sleep(time.Second)
    _, span := tracer.Start(
                r.Context(),
                "db-query",
            )

    // imagine this is a slow DB query.
    time.Sleep(time.Second * 3)
    span.End()
    _, _ = w.Write([]byte("Hello, World!"))
}
```

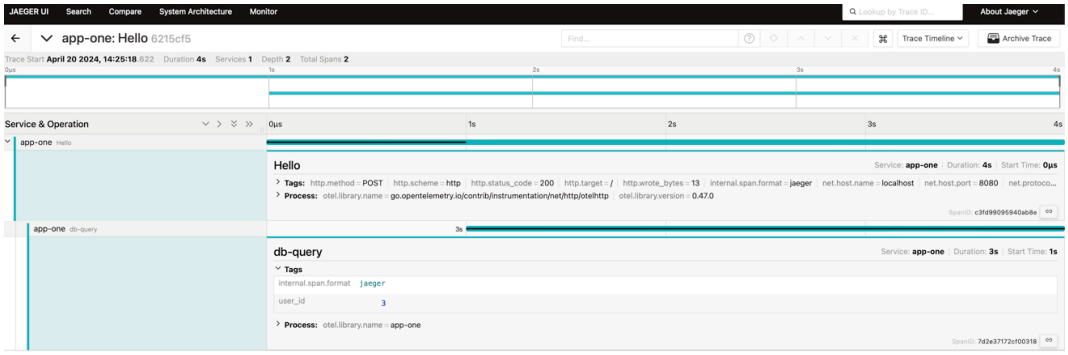
You can see a little clearer how this works.

ADDING ATTRIBUTES TO SPANS

As well as all the automatic instrumentation we have been getting for free so far, we can also add our own attributes to help us debug in the future. For example, perhaps we want to add the **`user_id`** as an attribute to our database queries:

```
func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    time.Sleep(time.Second)
    _, span := tracer.Start(
                r.Context(),
                "db-query”,
               )
    span.SetAttributes(attribut-
e.Int("user_id", 3))
    // imagine this is a slow DB query.
    time.Sleep(time.Second * 3)

    span.End()
    _, _ = w.Write([]byte("Hello, World!"))
}
```

Which yields a trace like:



As you can imagine, this can be exceptionally useful for understanding specific scenarios that are leading to performance issues or errors. Perhaps a certain user_id is much slower than all the others because they have thousands more products than any other customer on your platform and there is no index on your product column? I have seen this issue a few times.

CAPTURING ERRORS

Although you could do something like the below, there is a better way.

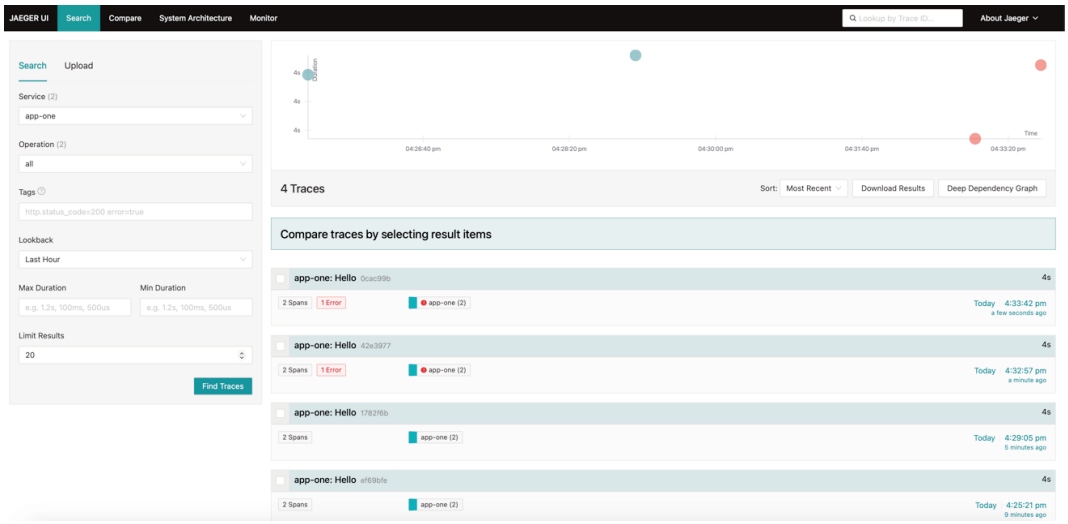
```
span.SetAttributes(attribute.String("error",
err.Error))
```

Let's update the code in our SimpleHandler one more time to represent a database error being returned. It now looks like this:

```
func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    time.Sleep(time.Second)
    _, span := tracer.Start(
                r.Context(),
                "db-query",
               )
    span.SetAttributes(attribut-
e.Int("user_id", 3))

    // imagine this is a slow DB query.
    time.Sleep(time.Second * 3)

    // imagine a service returned an error
    e := errors.New("db_commit_error")
    span.SetStatus(codes.Error, e.Error())
    span.End()
    _, _ = w.Write([]byte("Hello, World!"))
```

```
}
```

Now in the Jaeger UI, we can see at a glance error traces which is incredibly helpful!



OpenTelemetry does provide **`span.RecordError()`** too, but for some reason it does not automatically set the status to be error automatically. I therefore do not recommend using it.

TRACING BETWEEN DIFFERENT SERVICES

One of the most powerful features of distributed tracing is the ability to trace requests across multiple micro-services. OpenTelemetry automatically propagates trace context information through HTTP headers, allowing you to follow a request's journey through your entire distributed architecture.

In the Jaeger UI, you'll be able to see spans from different services seamlessly connected, providing end-to-end visibility into the request flow. This becomes particularly useful when diagnosing issues that span multiple components or services. Let's see an example of how to do that.

Firstly, let's write the two simplest Go applications that we can to demonstrate this behavior. Here is app-0ne in its entirety.

```
package main

import (
    "fmt"
    "go.opentelemetry.io/contrib/instrumenta-
tion/net/http/otelhttp"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/codes"
    "go.opentelemetry.io/otel/exporters/jaeger"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/sem-
conv/v1.7.0"
    trace2 "go.opentelemetry.io/otel/trace"
    "log"
    "net/http"
)

var tracer trace2.Tracer

func main() {
    exporter, err := jaeger.New(
        jaeger.WithCollectorEndpoint(),
    )
    if err != nil {
        log.Fatal(err)
    }

    // Create Trace Provider
    tp := trace.NewTracerProvider(
        trace.WithBatcher(exporter),
        trace.WithResource(
            resource.NewWithAttributes(
```

```
                semconv.SchemaURL,
                semconv.ServiceName-
Key.String("app-one"),
            ),
        ),
    )

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(
        propagation.TraceContext{},
    )
    tracer = tp.Tracer("app-one")

    http.Handle(
        "/",
        otelhttp.NewHandler(
            http.HandlerFunc(SimpleHandler),
            "Hello",
        ),
    )
    if err := http.ListenAnd-
Serve(":8080", nil); err != nil {
        log.Fatal(err)
    }
}

func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    ctx, span := tracer.Start(
        r.Context(),
        "calling app-two",
    )
    defer span.End()

    req, err := http.NewRequestWithContext(
```

```
        ctx,
        http.MethodGet,
        "http://localhost:8081/",
        nil,
    )
    if err != nil {
        span.SetStatus(
            codes.Error,
            err.Error(),
        )
        w.WriteHeader(http.StatusInternal-
ServerError)
        return
    }

    client := http.Client{
        Transport: otelhttp.NewTransport(http.De-
faultTransport),
    }
    res, err := client.Do(req)
    defer res.Body.Close()
    if err != nil {
        span.SetStatus(
            codes.Error,
            err.Error(),
        )
        w.WriteHeader(http.StatusInternal-
ServerError)
        return
    }
    if res.StatusCode != http.StatusOK {
        span.SetAttributes(
            attribute.Int("upstream_status_-
code", res.StatusCode),
        )
```

```
        w.WriteHeader(http.StatusInternal-
ServerError)
        return
    }
    fmt.Fprintf(w, "Hello from app-one!")
}
```

There's a few interesting lines I want to call out here. In my experience, Open Telemetry is particular about the configuration and missing any of the above lines could lead to it not working.

```
otel.SetTextMapPropagator(propagation.Trace-
Context{})
```

The line tells Open Telemetry to propagate our contexts over the wire(in practice, this means it adds HTTP headers to the outgoing request) . If you find that your traces are not propagating between services correctly, this one line missing might be the reason.

```
req, err := http.NewRequestWithContext(
    ctx,
    http.MethodGet,
    "http://localhost:8081/",
    nil
)
if err != nil {
    span.SetStatus(
        codes.Error,
        err.Error()
    )
    w.WriteHeader(http.StatusInternalServerError)
    return
}

client := http.Client{
```

```
    Transport: otelhttp.NewTransport(http.De-
faultTransport),
}
res, err := client.Do(req)
if err != nil {
    span.SetStatus(
        codes.Error,
        err.Error()
    )
    w.WriteHeader(http.StatusInternal-
ServerError)
    return
}
defer res.Body.Close()
```

In this line here we are creating a new http request and we are using the context that we started a couple of lines above. Again, if we do not do this, our trace will not propagate. It's also really important we use a http client wrapped in the **`Otel.NewTransport`** function or, you guessed it, it won't work!

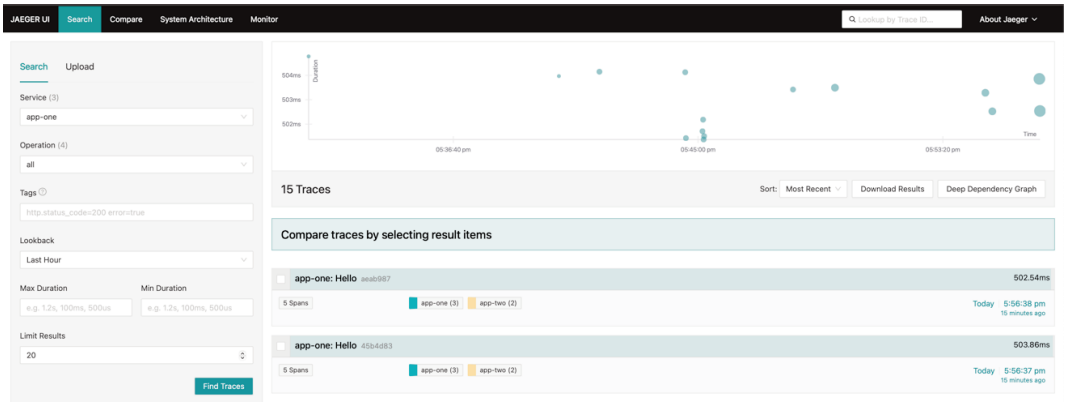
Thankfully app-two is really simple and doesn't contain anything we have not learnt yet. Here it is in its entirety.

```
package main

import (
    "fmt"
    "go.opentelemetry.io/contrib/instrumenta-
tion/net/http/otelhttp"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/jaeger"
    "go.opentelemetry.io/otel/propagation"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
```

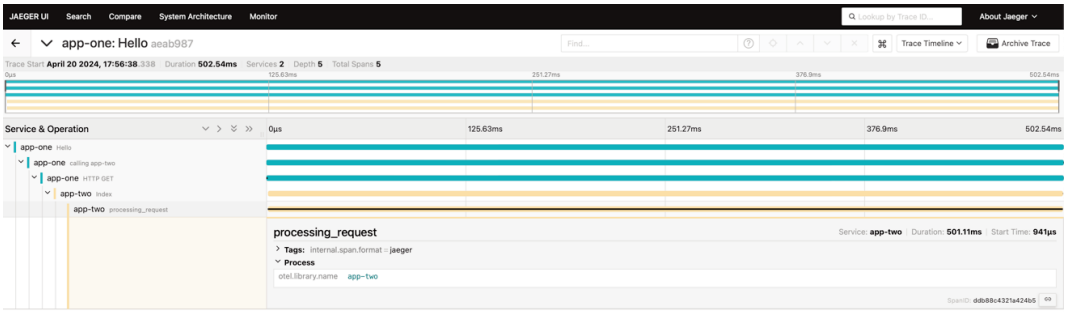
```
    semconv "go.opentelemetry.io/otel/sem-
conv/v1.7.0"
    trace2 "go.opentelemetry.io/otel/trace"
    "log"
    "net/http"
    "time"
)

var tracer trace2.Tracer

func main() {
    // Initialize Jaeger Exporter
    exporter, err := jaeger.New(
        jaeger.WithCollectorEndpoint(),
    )
    if err != nil {
        log.Fatal(err)
    }

    // Create Trace Provider
    tp := trace.NewTracerProvider(
        trace.WithBatcher(exporter),
        trace.WithResource(
            resource.NewWithAttributes(
                semconv.SchemaURL,
                semconv.ServiceName-
Key.String("app-two"),
            ),
        ),
    )

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(
        propagation.TraceContext{},
    )
```

```
    tracer = tp.Tracer("app-two")

    // Define HTTP server and routes
    http.Handle(
        "/",
        otelhttp.NewHandler(
            http.HandlerFunc(SimpleHandler),
            "Index",
        ),
    )
    log.Fatal(
        http.ListenAndServe(":8081", nil),
    )
}

func SimpleHandler(w http.Response-
Writer, r *http.Request) {
    _, span := tracer.Start(
        r.Context(),
        "processing_request",
    )
    defer span.End()
    time.Sleep(time.Millisecond * 500)
    // Perform some operations here
    fmt.Fprintf(
        w,
        "Hello from app-two!",
    )
}
```

If we run both of these applications, make a request to http://local host:8080/ and then check the Jaegar-UI, we should see the following:



Look how cool that is! We have made one request, but Jaegar can see that it spans two services (app-one and app-two). If we click on a trace it is even more interesting:

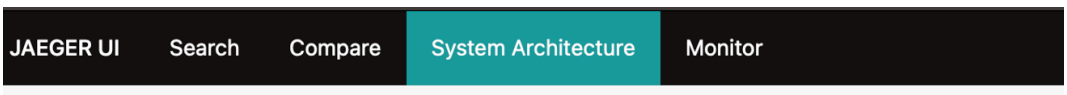


We can see a color-coded breakdown of our http request, where all the time was spent, any events that were attached to the trace, and any errors that may have occurred. Awesome!

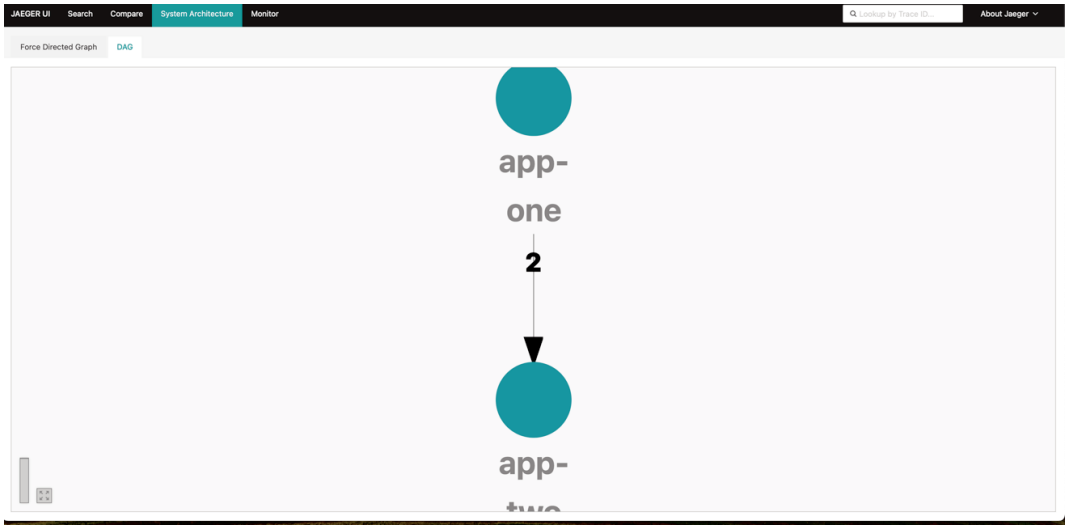
By doing this, we also unlocked another cool new feature which is the System Architecture tab in Jaegar.

AUTOMATIC DEPENDENCY MAPS

In Jaeger UI, if you click “System Architecture in the top menu:



You'll see:



This might not look very impressive, but take a step back and think about what just happened. Simply by making http calls between applications and propagating traces, we now have a real-time view of which services are talking to which.

Here's a more complex example:

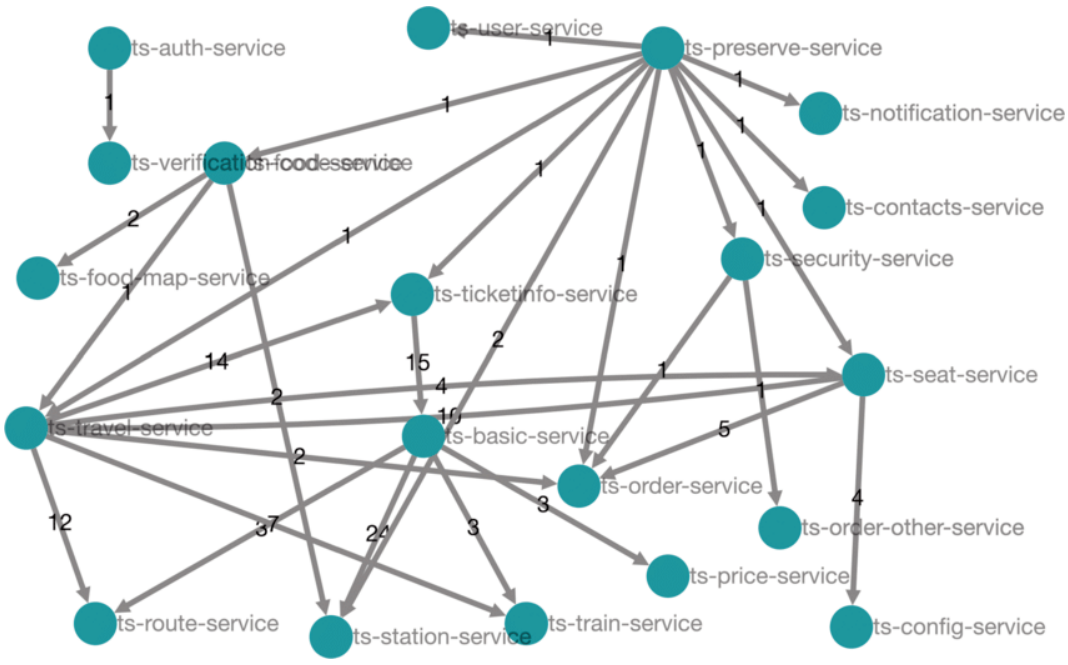


Image courtesy of researchgate.net

This is incredibly useful for debugging. Between the nodes on the graph, you can see the request count. This can help you understand which are the busiest services. Furthermore, you could use this to identify circular-dependencies as they appear.

TRACING BEYOND HTTP

Although the focus of the previous section, I want to be clear that the power of distributed tracing can extend far beyond HTTP calls between microservices. You can leverage this technique to trace and visualize various aspects of your application, including:

- Database calls
- gRPC communications
- Asynchronous messaging (e.g., Kafka, RabbitMQ)
- External API integrations
- Batch processing jobs

By creating spans around critical sections of your code, you can gain valuable insights into the performance and behavior of your entire system, enabling you to make informed decisions and optimize your applications for better user experiences.

YOUR TURN!

Take a look at the exercise package in this repo. You can find it in cmd/exercise.[4]

Firstly, run **`docker compose up`** and ensure you can get to the Jaegar UI on 16686.

If you run the binary and make a request to http://localhost:8081/, you should get a 200 response in a few seconds. This is much too slow and we must figure out why it is taking so long.

If you look in **`CalcHandler`**, we are doing two function calls.

Your challenge, using only tracing, is to do the following:

• Figure out if the functions are giving the same results.

• Figure out which of the function calls is slower. How much slower is it?

Once you have given it your best, you can see me step through the solution here[5]

WRAPPING UP WITH A WARNING

We covered a lot in this chapter and even though all of our implementations were in Go, what you have learnt is language agnostic.

Before we move on, I want to give a couple of warnings. Whilst it's tempting to capture every single trace from your system, doing so without a sampling strategy can lead to significant issues such as:

Performance Impact:

Tracing every request can add overhead to your system. This can slow down your application, especially under high load, potentially affecting user experience and system reliability.

Data Overload

Without sampling, you'll generate a massive volume of trace data. This can be overwhelming to analyze and may obscure important information. Too much data can make finding the needle in the haystack even harder.

Increased Costs

More data means higher storage and processing costs. Storing every trace, especially in large-scale systems can become prohibitively expensive very quickly.

Resource Saturation

Your tracing backend (in our example, Jaeger) and any associated databases will face increased load. This can lead to performance degradation, increased latency in trace retrieval, and even system outages in extreme cases.

To overcome these challenges, you can:

Implement a Sampling Strategy

Start with a sampling rate that makes sense for your system's load and the criticality of the data. Adaptive sampling, where the rate changes based on system load, can be particularly effective.

Focus on Key Transactions

Identify critical paths or transactions in your system and prioritize tracing them.

Monitor and Adjust

Continuously monitor the impact of tracing on your system. Be prepared to adjust your sampling strategy as your system scales.

Use Sampling for Learning

Initially, a higher sampling rate can be useful for understanding your system's behavior. Once you have this insight, reduce the sampling rate.

Remember, the goal of distributed tracing is to gain insights into system performance and issues, not to drown in data.

With that, I'll see you in the next chapter!

PROFILING & PPROF

Profiling is the process of measuring the time and space (memory) complexity of a program. It serves as a powerful tool to identify the parts of your code that consume the most resources; whether that is CPU, time or memory usage.

Like other things in this book, profiling isn't a Go-specific thing, but Go does have amazing first class support for it.

WHY PROFILE?

Profiling offers benefits for both debugging and for general optimization.

Identify Bottlenecks

By pinpointing the slow and resource-intensive parts of your code, you can focus your efforts on optimizing the right areas.

Optimize Performance

Profiling empowers you to make informed decisions, ensuring your application runs smoothly and efficiently.

Improve Resource Usage

Reduce your application's memory footprint and increase efficiency, especially in cloud environments where resources are billed.

Enhance User Experience

Ultimately, profiling allows you to provide a seamless and responsive experience for your users, ensuring they don't have to wait endlessly for responses.

As we discussed previously, the Go standard library offers built-in support for profiling, and they come in a few different flavors.

CPU Profile

The CPU profile measures the time spent on different functions or methods within your application. This profile is invaluable for identifying performance bottlenecks and optimizing CPU-bound operations.

Memory Profile

The memory profile tracks memory allocation and usage throughout your application's lifecycle. With this profile, you can pinpoint memory leaks and excessive allocations, ensuring your application maintains a lean and efficient memory footprint.

Concurrency Profile

In the world of concurrent programming, the concurrency profile is your ally. It analyzes the behavior of goroutines, threads, and other concurrency primitives, helping you identify potential race conditions, deadlocks, and other concurrency-related issues.

Now we know at a high level what profiling is. Let’s go ahead and add the profiler to a Go application.

ADDING PROFLING - THE SIMPLE WAY

If you're looking for the easiest way to add profiling capabilities to your Go application, even if your application is not a web application, the **`net/http/pprof`** package has got you covered. To add it, all you need to do is add this import:

```
import _ “net/http/pprof”
```

By simply importing this package as a side effect, you'll automatically add a profiling endpoint to your HTTP server.

Once you've added the **`net/http/pprof`** package, you can run the profiler using the **`go tool pprof`** command. For example, if your application is running on **`localhost:8080`**, you can run

```
go tool pprof http://localhost:8080/de-
bug/pprof/profile
```

This command will capture a CPU profile and open the pprof tool, allowing you to analyze the collected data.

No HTTP server? No Problem

What if your application doesn't have an HTTP server? pprof still has you covered. You can start a dedicated HTTP server solely for profiling purposes. Here's an example:

```
package main

import (
    "log"
    "net/http"
    _ "net/http/pprof"
)

func main() {
    log.Println(
     http.ListenAndServe(
      "localhost:6060",
       nil,
      ),
    )
}
```

More likely than not, your application will do more than just start pprof server. You probably want to have your own business logic which may include another http server. This is a good way to do that:

```
package main

import (
    "context"
```

```
    "log"
    "net/http"
    _ "net/http/pprof"
    "golang.org/x/sync/errgroup"
)

func main() {
    ctx := context.Background()
    g, ctx := errgroup.WithContext(ctx)

    g.Go(func() error {
        log.Println("Starting pprof server on
localhost:6060")
        return http.ListenAndServe(
                "localhost:6060",
                 nil,
               )
    })

    g.Go(func() error {
        log.Println("Starting main application
logic...")
        // Your main application logic here
        return nil
    })

    if err := g.Wait(); err != nil {
        log.Fatal(err)
    }
}
```

In this example, we start the profiling HTTP server on **localhost:6060** and import the **net/http/pprof** package for its side effects. Now, you can run the profiler against this dedicated server:

```
go tool pprof http://localhost:6060/debug/pro-
file/pprof
```

PROFLING WITHOUT SIDE EFFECTS

So far we have been getting our pprof server running by importing it like this:

```
_ "net/http/pprof"
```

Generally, importing with an _ is seen as an anti-pattern in Go [1]because it implies that the imported package is only being used for its side effects and not for any of its exported functions, types, or variables. This practice can make the code less readable and maintainable, as it is not immediately clear why the package is being imported. It can also obscure the actual dependencies of your code and lead to issues with initialization side effects, which might not be obvious to someone reading your code for the first time.

For example, importing a package solely for its side effects might make sense in certain situations (such as registering database drivers), but overuse or misuse can result in code that is harder to understand and maintain. It is generally better to import the package explicitly and use its exported functions to make your intentions clear.

Therefore to have more control over the debug endpoints and to avoid relying on side effects, you can explicitly register the pprof handlers. Here's an example:

```
package main

import (
    "net/http"
    "net/http/pprof"
)

func main() {
```

```
    mux := http.NewServeMux()

    mux.HandleFunc(
        "/debug/pprof/",
        pprof.Index
    )

    mux.HandleFunc(
        "/debug/pprof/cmdline",
        pprof.Cmdline
    )
    mux.HandleFunc(
        "/debug/pprof/profile",
        pprof.Profile
    )
    mux.HandleFunc(
        "/debug/pprof/symbol",
        pprof.Symbol
    )
    mux.HandleFunc(
        "/debug/pprof/trace",
        pprof.Trace
    )

    http.ListenAndServe(
        ":8080",
        mux
    )
}
```

This means you can also customize the path of the endpoints used.

PROFILING THE HEAP (MEMORY)

As Go developers, we often deal with dynamic memory structures such as slices, whose size cannot be determined at compile time. In such cases, Go allocates these structures on the heap, which can lead to memory issues and performance bottlenecks if not managed properly. Let's explore what the heap is in a little more detail and also how to optimize it in our programs for better performance.

What is the heap?

The heap is a region of memory where dynamically allocated objects reside. While the heap is our friend, allowing us to create objects that live beyond a single function call, it can also be a significant source of slowdown in Go programs because allocating and deallocating memory from the heap is an expensive operation. Furthermore, the heap remains allocated until it's explicitly freed by the garbage collector. While garbage collection is generally helpful, its cycles can introduce significant pauses in performance-critical applications.

Signals that you may need to optimize your memory usage include:

• High memory usage in your application

• Large garbage collection pauses

Both of these metrics can be exposed by Prometheus, which we discussed a couple of chapters ago.

A Basic Example

To illustrate the sort of problem that heap analysis might be useful for, let's consider a contrived example of an HTTP server running in a goroutine. Within this routine, we have a for loop that unnecessarily creates a slice on each iteration, multiplying its values by two:

```
package main

import (
```

```
    "fmt"
    "net/http"
    "net/http/pprof"
    "time"
)

func main() {
    go func() {
        fmt.Println(http.ListenAnd-
Serve(":8082", nil))
    }()

    for i := 0; i < 1000000; i++ {
    // Unnecessarily creat-
ing a slice for each iteration.
        data := make([]int, 100)
       simulateProcessing(data)
    }

    time.Sleep(time.Second * 20)
}

func simulateProcessing(data []int) []int {
    result := make([]int, len(data))
    for i, v := range data {
        result[i] = v * 2
    }
    return result
}
```

While this may seem like a silly example, it's not uncommon to encounter situations where data structures are created, used, and discarded within loops, leading to unnecessary memory allocations and potential performance issues.

If we run this program, it will hang for a while, as you'd expect. Whilst it's still running, we can open another terminal and run the following command:

```
go tool pprof http://localhost:8082/debug/
pprof/heap
```

You should see some text similar to the following:

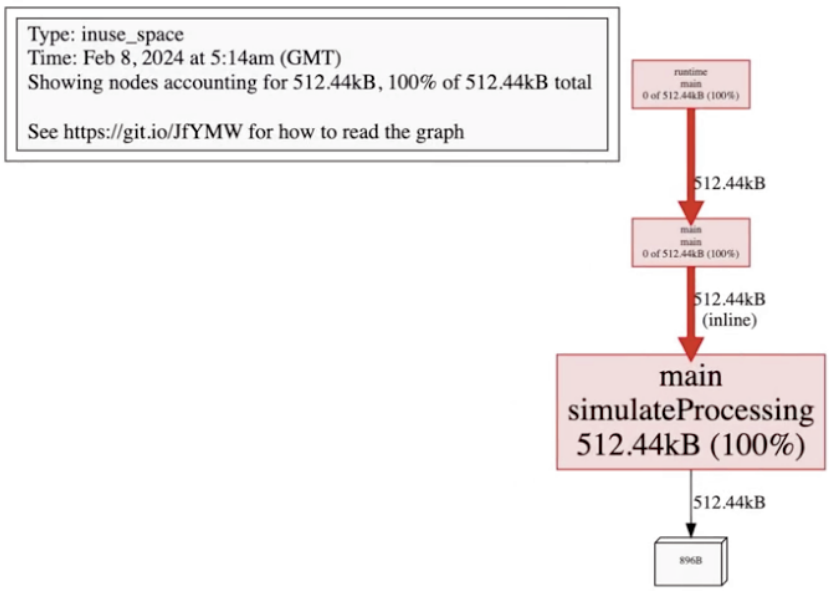
```
Fetching profile over HTTP from http://localhost:
8082/debug/pprof/heap

Saved profile in /Users/matthew-
boyle/pprof/pprof.alloc_objects.alloc_space.i-
nuse_objects.inuse_space.005.pb.gz

Type: inuse_space
Time: Feb 8, 2024 at 5:12am (GMT)

No samples were found with the default sample
value type.
Try "sample_index" command to analyze different
sample values.

Entering interactive mode (type "help" for
commands, "o" for options)

(pprof)
```

This means everything has worked and we now have an interactive pprof terminal open. We can prove this by typing the command ***top***. You'll see something like:

```
Entering interactive mode (type "help" for commands, "o" for options)

(pprof) top
```

```
Showing nodes accounting for 512.44kB, 100% of 512.44kB total
      flat  flat%   sum%        cum   cum%
  512.44kB  100%   100%   512.44kB  100%  main.simulateProcessing (inline)
         0    0%   100%   512.44kB  100%  main.main
         0    0%   100%   512.44kB  100%  runtime.main
(pprof)
```

This is showing us the functions with the most significant memory allocations. You can also type **web** and the tool will open a graph in your browser that looks something like this:



You can use this to look at varying paths through your application. This example is very simple, but they can get really big, which is a little overwhelming when just starting out.

A Real World Example

There has been a challenge going around the internet which has been a fun exploration of how far modern Java can be pushed for achieving one billion rows from a text file. However, we can also do it in Go!

There's an amazing write up of the full approach on the byteSizeGo blog[2].

In the challenge, you are given a text file containing temperature values for a range of weather stations. Each row is one measurement in the format *<string: station name>;<float: measurement>* and, as you probably guessed, there are a billion of these rows.

As an output, you must for each unique station, find the minimum, average and maximum temperature recorded and emit the final result on STDOUT in station name's alphabetical order with the format **{<station name>:<min>/<average>/<max>;<station name>:<min>/<average>/<max>}**.

There are some further constraints too:

• the temperature value is within [-99.9, 99.9] range.

• the temperature value only has exactly one fractional digit.

• the byte length of station name is within [1,100] range.

• there will be a maximum of 10,000 unique station names.

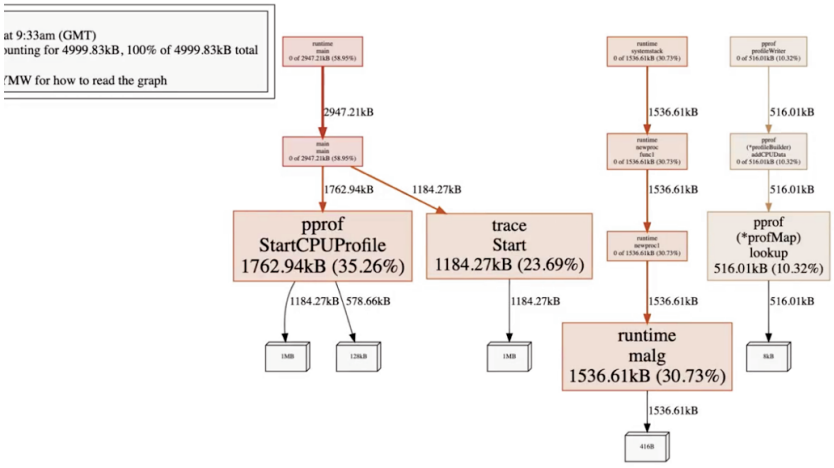
• rounding of temperature values must be done using the semantics of IEEE 754 rounding-direction "roundTowardPositive".

In the blog, the author Shraddha starts with a naive solution which takes ~6 minutes, and through profile-guided optimization[3] manages to get her solution down to 14 seconds.

Thankfully, Shraddha published all of her code and profiles to a Github repo[4]. This approach can be invaluable for open-source projects, where contributors can work together to optimize performance, or in enterprise environments, where profiles can be regularly captured and analyzed. By inspecting the profiles using pprof, we can identify bottlenecks and areas for improvement, just as Shraddha did.

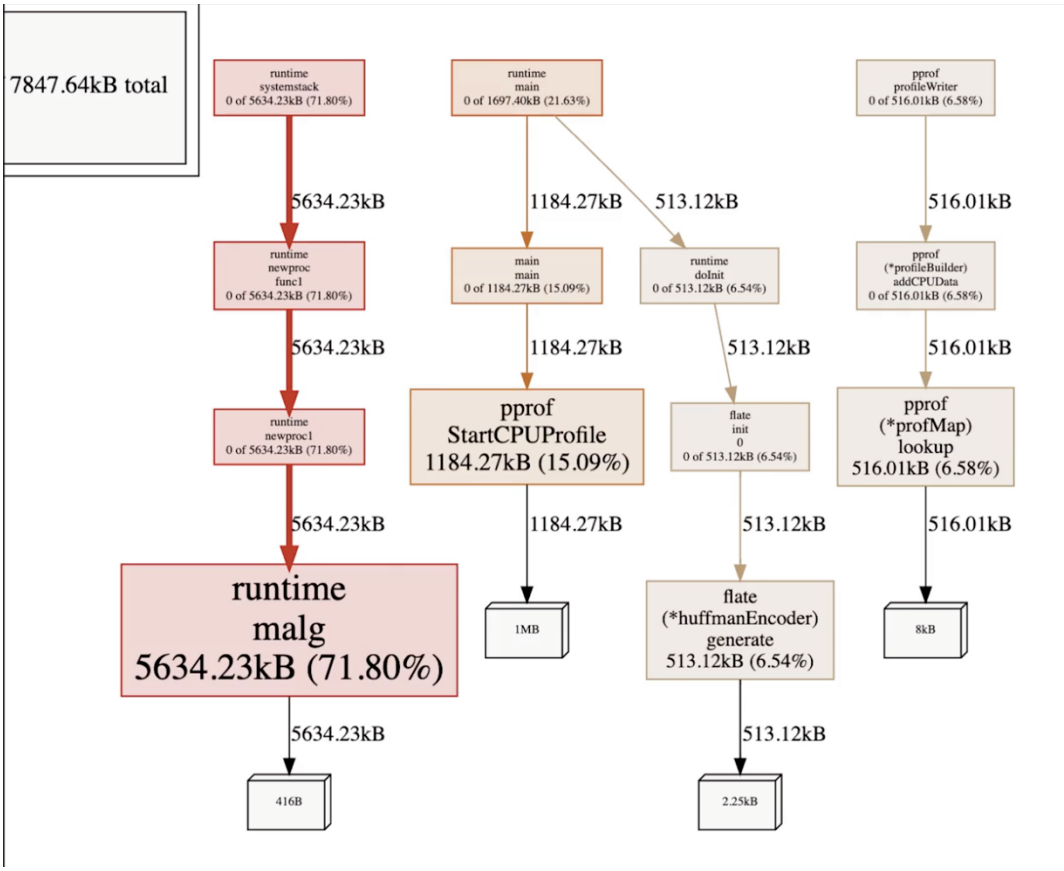
Once you have cloned the repo, you can run **go tool pprof mem-**

`1000-invert-prof` . If you then type **`web`**, you will see the following:



This has a lot more going on than our simple example. You'll see **`malg`** is taking up 30.73%, which is the thing responsible for handling dynamic memory allocation. You'll also see on the left **`StartCpu-Profile`** which is using 35.26%. This proves a very useful to know fact; profiling does have an overhead and we therefore do not want to leave it on all time, especially in performance critical systems.

Here's another example from the same repo:



If we ignore the **`malg`** and the **`startCPUProfile`** that we have already discussed, we can see that some memory is being used to do **`huffmanEncoder`** and **`lookup`**. If I was focusing on debugging my application to make it as fast as possible, that 6.54% of time we are spending dynamically allocating memory for **`huffmanEncoder`** would be something I would want to go after as I know it's generally an expensive process.

Back to Basics

Now we have looked at some pretty complex scenarios, lets go back to our basic example:

```
package main

import (
    "fmt"
    "net/http"
```

```
    _"net/http/pprof"
    "time"
)

func main() {
    go func() {
        fmt.Println(http.ListenAnd-
Serve(addr: ":8082", handler: nil))
    }()

    for i := 0; i < 1000000; i++ {
    // Unnecessarily creat-
ing a slice for each iteration.
        data := make([]int, 100)
       simulateProcessing(data)
    }

    time.Sleep(time.Second * 20)
}

func simulateProcessing(data []int) []int {
    result := make([]int, len(data))
    for i, v := range data {
        result[i] = v * 2
    }
    return result
}
```

In this example, removing dynamic memory allocation is really simple. We know exactly how big our slice needs to be, we can encourage the compiler to do some clever things to make our program be more efficient. That would looks like this.

```
package main
```

```
import (
    "fmt"
    "net/http"
    _"net/http/pprof"
    "time"
)

func main() {
    go func() {
        fmt.Println(
            http.ListenAndServe(
                addr: ":8082",
                handler: nil
            )
        )
    }()

    for i := 0; i < 1000000; i++ {
        data := make([]int, 100)
        simulateProcessing(data)
    }

    time.Sleep(
        time.Second * 20
    )
}

func simulateProcessing(data []int) []int {
    result := make([]int, len(data))
    for i, v := range data {
        result[i] = v * 2
    }
    return result
}
```

Sometimes it really is about making the small changes!

CPU TRACING AND PPROF LIST

In the world of software development, efficiency is a critical factor that can make or break an application's performance. While various components contribute to a program's overall efficiency, the CPU is often the bottleneck, especially when it comes to computationally intensive tasks. Processing calculations can be a time-consuming endeavor, taking seconds, minutes, or even hours in some cases.

If you're aiming to enhance your application's efficiency, optimizing CPU usage should be a top priority. In this section, we'll explore how to identify CPU-related issues and dive into strategies to debug and optimize your Go programs for improved performance.

The Fibonacci Sequence

If you have had the unfortunate need to go through a Leetcode style interview process, you have likely written a fibonacci function such as:

```
func Fib(n int) int {
    if n <= 1 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

This function finds the nth Fibonacci number. It calls itself with the two previous numbers. As the number n gets bigger, the time to finish grows quickly.

Let's run this code a few times with different inputs. For example, for **`fib(10)`** it would look as follows.

```
func main() {
```

```
    start := time.Now()
    fib(10)
    elapsed := time.Since(start)
    fmt.Printf("took %s", elapsed)
}

func fib(n int) int {
    if n <= 1 {
        return n
    }
    return fib(n-1) + fib(n-2)
}
```

if we run the fibonacci function with different inputs, we see that the runtime increases exponentially:

- Fib(1): The function completes almost instantly, taking mere microseconds.
- Fib(5): Still reasonably fast, but you might notice a slight delay.
- Fib(10): The computation time is starting to become noticeable, but it's still relatively quick.
- Fib(20): Now we're venturing into millisecond territory.
- Fib(40): Seconds start to tick by as the calculation takes longer to complete.
- Fib(60): Depending on your system's resources, this computation could take minutes or even hours to finish!

As you can see, even a simple algorithm like the Fibonacci sequence can quickly become a performance bottleneck as the input size increases.

To identify and optimize CPU-related issues in your Go programs, you'll need to use profiling tools. Go provides a powerful built-in profiling package called pprof (short for "performance profiler"),

which can help you pinpoint the areas of your code that are consuming the most CPU resources.

Let's add pprof and the http server back to our application as before. Your imports and main function should now look as follows:

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
    "time"
)

func main() {
    start := time.Now()
    go func() {
        http.ListenAndServe(":8082", nil)
    }()
    fib(60)
    elapsed := time.Since(start)
    fmt.Printf("took %s", elapsed)
}
```

Start the program, and in a separate terminal window run:

```
curl http://localhost:8082/debug/pprof/profile?
seconds=30 > cpu.prof
```

This command uses cURL to request a CPU profile from your running Go application. The **`profile?seconds=30`** query parameter tells pprof to collect 30 seconds worth of CPU profiling data, which is then saved to a file named cpu.prof.

Once the profile has been captured, we can analyze it using the pprof tool by running ***`go tool pprof cpu.prof`***. As before, you should see something similar to:

```
Type: cpu
Time: Feb 9, 2024 at 6:51pm (GMT)
Duration: 30.13s, Total samples = 27.23s (90.37%)
Entering interactive mode (type "help" for commands, "o" for options)
```

It gives us some useful metadata here, and confirms the type of the profile, which is always useful! As before, we can type ***top*** and you'll see the following:

```
(pprof) top
Showing nodes accounting for 27.14s, 99.67% of 27.23s total
Dropped 25 nodes (cum <= 0.14s)
flat  flat%  sum%        cum   cum%
27.14s 99.67% 99.67%     27.17s 99.78%  main.fib
0     0%     99.67%     27.17s 99.78%  main.main
0     0%     99.67%     27.17s 99.78%  runtime.main
```

Now we are a little bit more familiar with the tool, let's go through the output line by line.

- ***`(pprof) top`***: This line indicates that the top command within the pprof tool was executed. The top command is used to display the functions where the most time is spent.
- ***`Showing nodes accounting for 27.14s, 99.67% of 27.23s total:`*** This tells us that the profiling data being shown accounts for 27.14 seconds of CPU time, which is 99.67% of the total CPU time sampled (27.23 seconds).
- ***`Dropped 25 nodes (cum <= 0.14s):`*** This indicates that 25 nodes (functions or methods) have been excluded from this output because their cumulative time is less than or equal

to 0.14 seconds, suggesting they are not significant contributors to the total CPU time.

The table below these lines provides detailed information on CPU time consumption:

- ***flat:*** The amount of CPU time spent in the function itself.
- ***flat%:*** The percentage of the total CPU time spent in the function itself.
- ***sum%:*** The cumulative percentage of the CPU time accounted for by this function and all functions listed above it.
- ***cum:*** The cumulative CPU time spent in this function plus all functions that it called.
- ***cum%:*** The cumulative percentage of the total CPU time accounted for by this function and all functions that called it.

The actual data in the table lists specific functions:

- ***main.fib:*** This function has consumed 27.14 seconds of CPU time, which is 99.67% of the total time sampled, both in flat and cumulative terms.
- ***main.main and runtime.main***: These entries have 0 seconds listed under flat, meaning they did not directly consume CPU time. However, because main.fib is called from these functions, they show the cumulative time (cum) as 27.17 seconds, which is a slight over-summing from the main.fib function, possibly due to measurement granularity or other overhead.

Overall, this output shows that the main.fib function is the primary consumer of CPU resources and is likely the first place to look when optimizing the program for CPU usage. This is very unsurprising since we intentionally called our slow fib function from ***main()***, this is unsurprising. What might be more interesting is *where* exactly in

the function that time is being spent. It's time to introduce a new command.

pprof list

In our terminal, enter **`pprof list main.fib`**. You should see the following:

```
Total: 27.23s
ROUTINE ======================== main.fib in /Users/matthewboyle/Dev/ultimate-debug-
ging-with-go-profiling/cmd/main.go
    27.14s  39.98s (flat, cum) 146.82% of Total
     9.46s  9.46s     22:func fib(n int) int {
     2.21s  2.22s     23: if n <= 1 {
     1.13s  1.13s     24:       return n
        .      .     25:   }
        .      .     26:
    14.34s  27.17s    27:   return fib(n-1) + fib(n-2)
        .      .     28:}
(pprof)
```

There's a lot we can learn from these lines, so let's go through them line by line again.

• ***Total: 27.23s:*** This indicates the total CPU time that was profiled.

• **`ROUTINE ======================== main.fib in /Users/matthewboyle/Dev/ultimate-debugging-with-go-profiling/cmd/main.go`**: This line specifies that the output is for the main.fib function, and it provides the file path where this function is located.

The table below includes:

• **`27.14s 39.98s (flat, cum) 146.82% of Total`**: These numbers provide the profiling statistics for the main.fib function.

• **`27.14s`** under flat is the direct CPU time spent in main.fib.

• **`39.98s`** under cum is the cumulative CPU time including all calls made by main.fib.

• **`146.82% of Total`** indicates that the cumulative time spent in this function and the functions it calls exceeds the total profiled time. This can happen when multiple instances of main.fib run concurrently (e.g., if the program is multi-threaded). This might seem confusing, since we did not spawn any additional goroutines, how is the total above 100%? This is because even though we did not spawn goroutines, the go runtime and scheduler can still execute functions concurrently. [5]

• ***`The line 9.46s 9.46s 22:func fib(n int) int {`*** shows the function signature line, with the CPU time directly spent on that line of code.

The lines:

• ***`2.21s 2.22s 23: if n <= 1 {`***

• ***`1.13s 1.13s 24: return n`***

show the amount of time spent executing each particular line within the fib function.

• The line **`14.34s 27.17s 27: return fib(n-1) + fib(n-2)`** represents the recursive call to main.fib, which consumes most of the CPU time.

• The dots . represent lines in the source code that didn't consume a measurable amount of CPU time.

We can learn from this we are spending a lot of time on the recursive calculation. Let's look at a strategy called memoization to see if we can make our code spend less time calculating and therefore overall be faster.

Memoization

Memoization is a technique used to speed up programs by storing the results of expensive function calls and reusing them when the same inputs occur again. This can save a lot of time and compute, especially for functions that are called repeatedly with the same inputs. It sounds like exactly the solution we need!

Here is how we might memoize this function.

```
var cache = make(map[int]int)

func fibCache(n int) int {
    if n <= 1 {
        return n
    }
    if val, found := cache[n]; found {
        return val
    }
    cache[n] = fibCache(n-1) + fibCache(n-2)
    return cache[n]
}
```

If we run this new function with **`fib(60)`**, which previously was taking close to an hour to complete for me:

```
func main() {
    start := time.Now()
    fib(10)
    elapsed := time.Since(start)
    fmt.Printf("took %s", elapsed)
}
```

It now completes in 54 microseconds! It's important to note that this optimization trades CPU time for memory usage. The cache can grow

quite large for larger input values, potentially leading to high memory usage. As with any optimization, it's crucial to strike a balance between CPU and memory use based on your application's specific requirements. For example, consider a scenario where your application is running on a resource-constrained environment, such as a server with limited memory or a containerized environment with strict resource limits. In such cases, optimizing for CPU performance alone may not be enough; you'll also need to consider the memory footprint of your application too. However, for our application, I think we can agree this trade-off is more than worth it.

Go provides various tools and techniques to help you monitor and analyze both CPU and memory usage, allowing you to strike the right balance for your specific use case. For example, you can use the pprof tool like we did above or you can use some Prometheus metrics like we explored earlier in the book.

PROFILING GOROUTINE USAGE

Goroutines are how concurrency is handled in the Go programming language. Goroutines are lightweight compared to threads, which are used in many other programming languages as their primary method of achieving concurrency, and you'll see a lot of people reach for them very quickly to "speed up" their program. However, an excessive amount of goroutines can still lead to issues in your application because context switching has an overhead. By context switching, I mean the operating system switching between these goroutines and threads. The go scheduler does a really good job making this as efficient as possible, but it still does take time.

More importantly, for the purpose of this book, using lots of goroutines can make an application harder to debug. Earlier in this book we annotated our goroutines in an attempt to make this a little better. Thankfully, there is another tool which makes this even easier.

Let's look at this simple Go application that is meant to represent someone spinning up a bunch of goroutines to "do work"

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
    "sync"
    "time"
)

func performTask(i int, wg *sync.WaitGroup) {
    defer wg.Done()
    // Simulate a task that doesn't necessari-
ly need a separate goroutine
    time.Sleep(10 * time.Millisecond)
    fmt.Printf("Task %d completed\n", i)
}

func main() {
    go func() {
        fmt.Println(http.ListenAnd-
Serve(":8082", nil))
    }()

    startTime := time.Now()
    var wg sync.WaitGroup
    numberOfTasks := 1000

    for i := 0; i < numberOfTasks; i++ {
        wg.Add(1)
        go performTask(i, &wg)
    }
```

```
    wg.Wait()
    fmt.Printf("All tasks complet-
ed in %s\n", time.Since(startTime))
}
```

We have 1000 tasks to complete, and we spin off a goroutine to complete that task. Once the task is complete, we call **`wg.Done()`** on it. If I run this program on my machine, it completes in around 15ms. This is pretty good and proves that goroutines really do make our program faster; in this program I had a 10ms wait in each goroutine so if the tasks were not handled concurrently it would have taken 10 seconds (1000 * 10ms) to complete.

Before we do some profiling, to make our life easier let's increase the wait time from 10 milliseconds to 10 seconds. The program now looks like this:

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
    "sync"
    "time"
)

func performTask(i int, wg *sync.WaitGroup) {
    defer wg.Done()
    // Simulate a task that doesn't necessari-
ly need a separate goroutine
    time.Sleep(10 * time.Second)
    fmt.Printf("Task %d completed\n", i)
}
```

```
func main() {
    go func() {
        fmt.Println(http.ListenAnd-
Serve(":8082", nil))
    }()

    startTime := time.Now()
    var wg sync.WaitGroup
    numberOfTasks := 1000

    for i := 0; i < numberOfTasks; i++ {
        wg.Add(1)
        go performTask(i, &wg)
    }

    wg.Wait()
    fmt.Printf("All tasks complet-
ed in %s\n", time.Since(startTime))
}
```

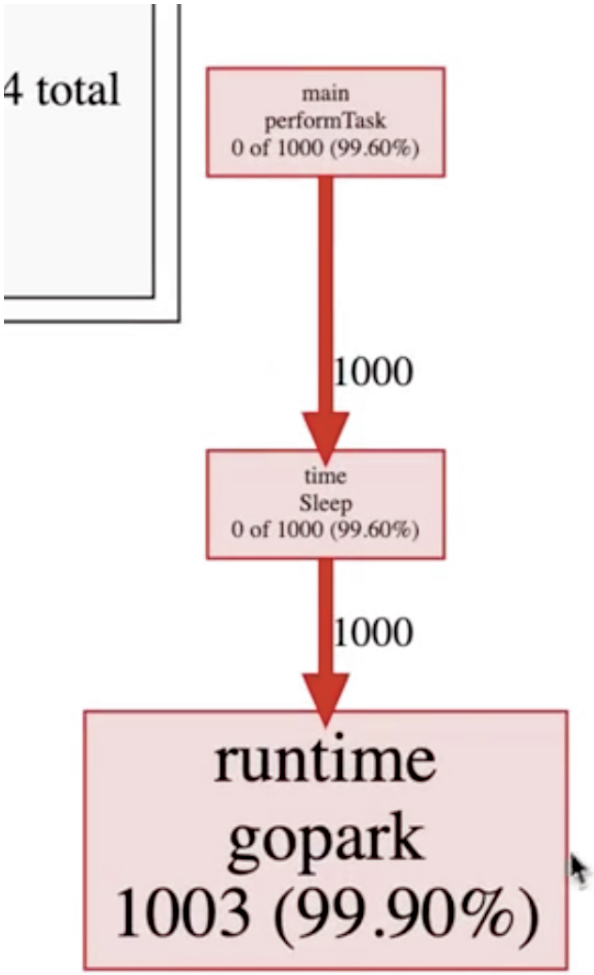
We can run a pprof command that profiles goroutine usage specifically. It looks like this:

```
go tool pprof http://localhost:8082/debug/pprof/
goroutine
```

This will open the familiar pprof terminal. As before we can use the top command. You should see an output such as:

```
Showing nodes accounting for 1003, 99.90% of 1004 total
Dropped 33 nodes (cum <= 5)
flat  flat%  sum%  cum  cum%
1003  99.90% 99.90% 1003  99.90% runtime.gopark
0     0%     99.90% 1000  99.60% main.performTask
0     0%     99.90% 1000  99.60% time.Sleep
(pprof)
```

This shows us we are using 1003 goroutines in total, and 1000 are being used by `performTask`. We can also use the command **`list`**, and it will show us that at the time of profiling, our goroutines are spending most of their time on the **`time.Sleep()`** which makes sense. Finally, we can run web as before, which shows us:



For the simple program we have here, perhaps the performance is ok. However, as your application gets more complex and performance more critical, I have found that experimenting with the amount of goroutines you use can have a meaningful impact on performance. Let's see this in action. Here is a refactor to the above code to only use 10 goroutines. I also lowered the wait time to 10ms:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func performTask(i int) {
    // Simulated task
    time.Sleep(10 * time.Millisecond)
    fmt.Printf("Task %d completed\n", i)
}

func worker(tasks chan int, wg *sync.WaitGroup) {
    for task := range tasks {
        performTask(task)
        wg.Done()
    }
}

func main() {
    startTime := time.Now()
    var wg sync.WaitGroup
    numberOfTasks := 1000
    numberOfWorkers := 10
    tasks := make(chan int, numberOfTasks)

    for i := 0; i < numberOfWorkers; i++ {
        go worker(tasks, &wg)
    }

    for i := 0; i < numberOfTasks; i++ {
        wg.Add(1)
        tasks <- i
```

```
    }

    wg.Wait()
    close(tasks)
    fmt.Printf("All tasks complet-
ed in %s\n", time.Since(startTime))
}
```

If we run this, it takes around 1 second to complete. Which is much slower than before. However, if we profile it, the new application only uses 13 goroutines as you can probably predict. Considering that goroutines can use more memory due to context switching and potentially saturating resources, this *could* be considered an improvement. As you have realized at this point, optimizing applications is all about trade-offs. You often have to pick between using more CPU, more memory or increasing time whilst occasionally being able to optimize all three at the same time. Context matters; and it is going to depend on your application, where it is running and what you're trying to achieve.

EXERCISE

Take a look at the code here[6]. The README contains instructions on how it works and how to get it running.

Our goal is to get munge running on level 2 with 1million_passwords.txt in less than 30 seconds.

Here are the Go program arguments to run it if you are using an IDE.

```
-i "wordlists/1million_passwords.txt" -l 2 -o
"out.txt"
```

Try to do this without looking at the code first; our goal is to look at the profiles and take action based on what we see there.

Good luck!

Thank you to Nino Stephen [7] on Twitter for sharing this repo with me and approving it for this purpose.

Once you have given it your best, you can watch me give a solution to this here[8].

WRAPPING UP - SHOULD I JUST LEAVE PPROF RUNNING ALL THE TIME?

Running the pprof profiler continuously in a Go application might seem like a good idea for ongoing performance monitoring, but there are several reasons why it's not advisable:

- **Performance Overhead:** While pprof is designed to be efficient, it still introduces some overhead. Continuously profiling a production system can lead to increased CPU usage and memory consumption. This overhead, albeit small in short bursts, can accumulate over time and impact the application's overall performance, especially under high load. We can see this usage in some of the profiles we generate in this course.
- **Data Overwhelm:** pprof generates detailed profiling data, which can be incredibly valuable for pinpointing performance bottlenecks. However, continuously running it means you'll accumulate a vast amount of data.
- **Resource Consumption:** Profiling continuously requires resources not just on the application side but also in terms of storage and processing power to handle the generated data.
- **Security Considerations:** Exposing profiling endpoints in a production environment can pose security risks. If not properly secured, attackers could potentially access sensitive application performance data or exploit the profiling service

to cause denial of service. Generally, this shouldn't be exposed publically.

Remember to profile safely!

THANK YOU

If you have read this far, thank you so much for reading this book and I hope you learnt a lot. If you did, please tweet about it on X or give me your thoughts directly by emailing hello@bytesizego.com.

Speaking of bytesizego.com, there is a growing catalog of courses and other books being made available there - some of them free, some of them paid. Please check them out and send me an email if you have

any ideas for content you'd like to see, or even would like to make a course yourself!

I also want to offer my heartfelt thanks to Toni, Michael, Ansar, Swastik and Bartłomiej for being amazing technical reviewers. I could not have written this without you.

Thanks for reading.

- Matt Boyle

NOTES

CONTRIBUTORS

1. https://x.com/MattJamesBoyle
2. https://x.com/anfragment
3. https://www.linkedin.com/in/tonilovejoy
4. https://vlang.io/
5. http://twitter.com/micvbang

1. WELCOME!

1. https://twitter.com/mattjamesboyle

1. DEBUGGING BY EYE

1. https://gobyexample.com/interfaces
2. https://go.dev/tour/concurrency/11
3. https://go.dev/doc/effective_go
4. https://github.com/uber-go/guide/blob/master/style.md
5. https://golangci-lint.run/
6. https://goplay.tools/snippet/X2XUK2wnRPH
7. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2357585-methods-of-debugging-debugging-by-eye/7530834-exercise-solution

2. PAIR PROGRAMMING

1. https://www.jetbrains.com/help/go/code-with-me.html

3. LOGGING

1. https://pkg.go.dev/fmt#hdr-Printing
2. https://go.dev/blog/slog
3. https://github.com/uber-go/zap
4. https://www.elastic.co/
5. https://www.elastic.co/kibana
6. http://bytesizego.com/the-ultimate-guide-to-debugging-with-go
7. https://docs.docker.com/get-docker/
8. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2359216-methods-of-debugging-logging/7571287-exercise-solution

9. https://twitter.com/MattJamesBoyle/status/1746213913757196710

4. THE DEBUGGER

1. https://www.jetbrains.com/go/
2. https://code.visualstudio.com/
3. https://marketplace.visualstudio.com/items?itemName=golang.go
4. https://go.dev/doc/faq#no_goroutine_id
5. https://quii.gitbook.io/learn-go-with-tests
6. https://www.microsoft.com/en-us/research/wp-content/uploads/2009/10/Realizing-Quality-Improvement-Through-Test-Driven-Development-Results-and-Experiences-of-Four-Industrial-Teams-nagappan_tdd.pdf
7. https://github.com/MatthewJamesBoyle/ultimate-debugging-course-debug-module.
8. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2359222-methods-of-debugging-the-debugger/7608312-exercise-solution

1. METRICS

1. http://prometheus.io
2. https://www.bytesizego.com/blog/keeping-alive-with-go
3. https://prometheus.io/docs/guides/go-application/
4. https://www.bytesizego.com/blog/keeping-alive-with-go
5. https://stackoverflow.com/questions/51146578/what-use-cases-really-make-prometheuss-summary-metrics-type-necessary-unique
6. https://aws.amazon.com/prometheus/
7. https://cloud.google.com/stackdriver/docs/managed-prometheus
8. https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/prometheus-metrics-overview
9. https://github.com/MatthewJamesBoyle/ultimate-debugging-with-go-metrics/
10. https://promlabs.com/promql-cheat-sheet/
11. https://grafana.com/docs/grafana/latest/alerting/
12. https://prometheus.io/docs/alerting/latest/overview/
13. https://github.com/MatthewJamesBoyle/ultimate-debugging-with-go-metrics
14. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2359231-debugging-in-production-metrics/7659150-exercise-solution

2. DISTRIBUTED TRACING

1. https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
2. https://github.com/MatthewJamesBoyle/ultimate-debugging-with-go-tracing
3. https://github.com/MatthewJamesBoyle/ultimate-debugging-with-go-tracing/blob/main/docker-compose.yaml
4. https://github.com/MatthewJamesBoyle/ultimate-debugging-with-go-tracing

5. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2359239-debugging-in-production-tracing/7683889-exercise-solution

3. PROFILING & PPROF

1. https://dave.cheney.net/practical-go/presentations/gophercon-israel.html
2. https://www.bytesizego.com/blog/one-billion-row-challenge-go
3. https://go.dev/doc/pgo
4. https://github.com/shraddhaag/1brc
5. https://go.dev/blog/pprof
6. https://github.com/MatthewJamesBoyle/munge.
7. https://twitter.com/_ninostephen_
8. https://www.bytesizego.com/view/courses/the-ultimate-guide-to-debugging-with-go/2359245-debugging-in-production-profiling-pprof/7753649-exercise-solution