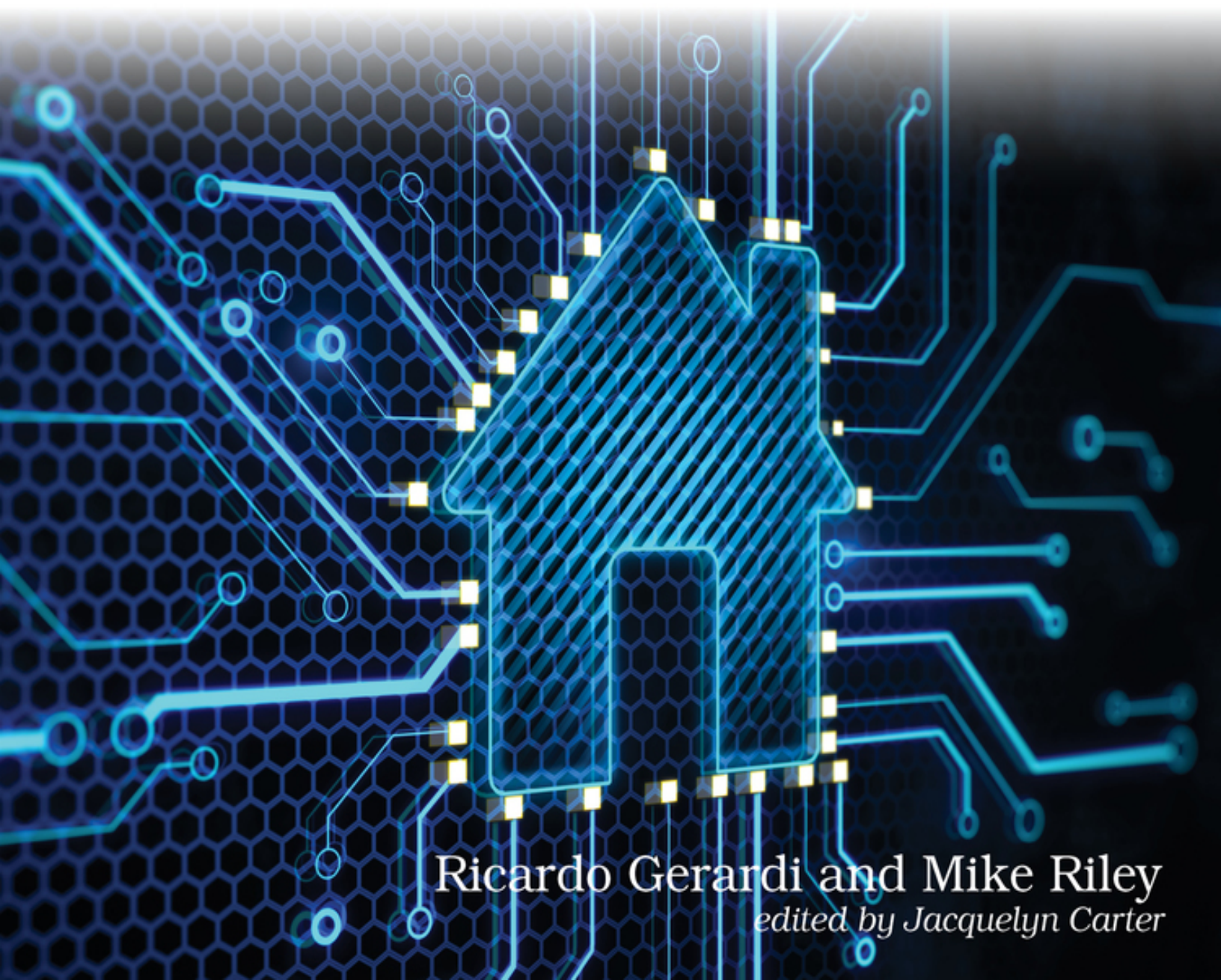


Automate Your Home Using Go

Build a Personal Data Center with Raspberry Pi, Docker, Prometheus, and Grafana

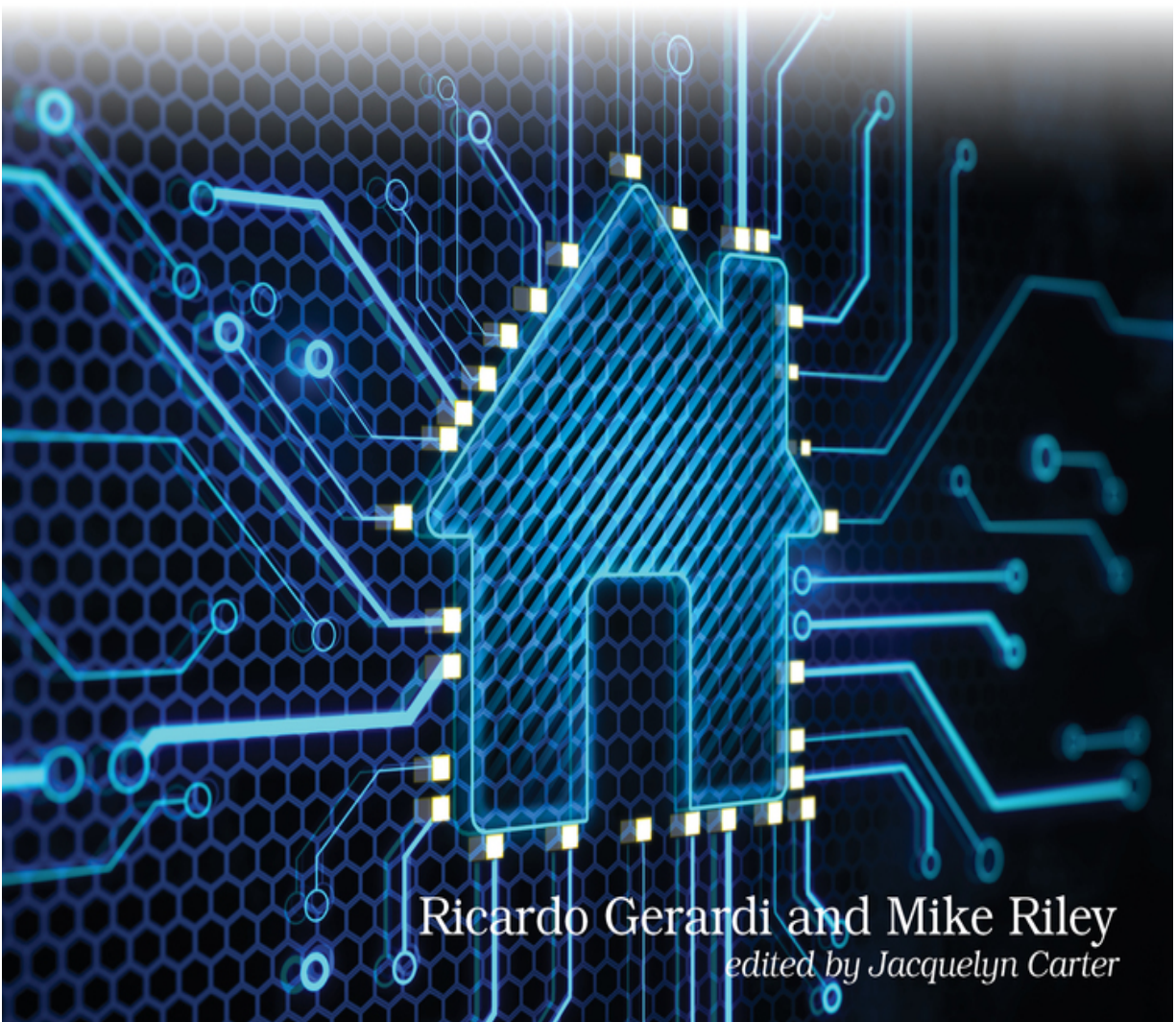


Ricardo Gerardi and Mike Riley
edited by Jacquelyn Carter

The
Pragmatic
Programmers

Automate Your Home Using Go

Build a Personal Data Center
with Raspberry Pi, Docker,
Prometheus, and Grafana



Ricardo Gerardi and Mike Riley
edited by Jacquelyn Carter

Automate Your Home Using Go

Build a Personal Data Center with Raspberry Pi, Docker, Prometheus, and Grafana

by Ricardo Gerardi, Mike Riley

Version: P1.0 (August 2024)

Copyright © 2024 The Pragmatic Programmers, LLC. This book is licensed to the individual who purchased it. We don't copy-protect it because that would limit your ability to use it for your own purposes. Please don't break this trust—you can use this across all of your devices but please do not share this copy with other members of your team, with friends, or via file sharing services. Thanks.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://pragprog.com>.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free

updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's policies. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to <https://pragprog.com/book/gohome>, the book's homepage.

Thanks for your continued support,

The Pragmatic Bookshelf

The team that produced this book includes: Dave Thomas (Publisher), Janet Furlow (COO), Susannah Davidson (Executive Editor), Jacquelyn Carter (Development Editor), Vanya Wryter (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

To my parents Fatima and Wilson who taught me how to be a kid.

To my kids Gisele, Livia, Elena, and Alice who taught me how to be a parent.

Ricardo

To my wife Marinette who has endured me for over 40 years, my kids Mitchell and Marielle for a little less, and my brother Frank who tops them all (are we really that old, bro?). I couldn't have been blessed with a better family.

Mike

Table of Contents

1. **Acknowledgments**

2. **Introduction**

1. Who This Book Is For

2. What's in This Book

3. About the Hardware

4. About the Code

5. Online Resources

3. **Part I. Setup**

1. **Getting Started**

1. Your Personal Data Center

2. Selecting a Raspberry Pi

3. Adding Other Hardware Components

4. Configuring the Software

5. Picking a Code Editor

6. Next Steps

2. **Building a REST API Server**

1. [Writing the Code](#)
 2. [Containing the Server](#)
 3. [Next Steps](#)
-
3. **3. [Deploying Your Personal Data Center](#)**
 1. [Deploying Applications with Linux Containers](#)
 2. [Managing Source Code with Gitea](#)
 3. [Monitoring and Alerting with Prometheus](#)
 4. [Visualizing Data with Grafana](#)
 5. [Next Steps](#)
-

4. **[Part II. Projects](#)**

1. **4. [Networking a Temperature Monitor](#)**
 1. [Understanding the Pico W Device](#)
 2. [Polling the Temperature](#)
 3. [Connecting the Pico W to Wi-Fi](#)
 4. [Creating the Pico W REST Server](#)
 5. [Creating the Prometheus Exporter](#)
 6. [Containing and Deploying the Exporter](#)
 7. [Creating the Grafana Dashboard](#)

8. [Next Steps](#)

2. **5. [Checking the \(Garage\) Door](#)**

1. [Understanding the GPIO](#)
2. [Wiring the Magnetic Switch to the GPIO](#)
3. [Coding the Magnetic Switch](#)
4. [Sending Notifications](#)
5. [Writing the Software](#)
6. [Configuring and Testing the Application](#)
7. [Containerizing the Deployment](#)
8. [Next Steps](#)

3. **6. [Lighting the Weather](#)**

1. [Polling the Weather](#)
2. [Changing the Color](#)
3. [Putting It All Together](#)
4. [Configuring the Application Settings](#)
5. [Containerizing and Deploying the App](#)
6. [Next Steps](#)

4. **7. [Watching the Birds](#)**

1. [Setting Up the Camera and InfraRed Sensor](#)

2. [Writing the Software](#)
3. [Sending Motion Notifications](#)
4. [Containing the Application](#)
5. [Configuring the Bird Feeder](#)
6. [Next Steps](#)

5. **8. [Go Build](#)**

1. [Designing Additional Projects](#)
2. [Expanding the Technologies](#)
3. [Improving Security](#)
4. [Advancing Electronics](#)
5. [Having Fun](#)

Early Praise for *Automate Your Home Using Go*

A fantastic introduction to home automation programming!

→ Doug Clarke

Cloud Engineering, Automation Specialist, Bell Canada

This book is impressive for its breadth and depth; projects are exciting and varied with implementations that dive into core technologies such as containers, graphing, configurations, not to mention the different hardware used. This is NOT your little Pi project book. It's much better!

→Mike Bengtson

CTO, Tack så Mycket

This book is a great resource for anyone looking to blend the power of home automation with the efficiency of the Go programming language. It comes full of practical insights and hands-on projects, making complex concepts accessible and exciting. Ricardo and Mike did a great job!

→Mihalis Tsoukalos

Author of *Mastering Go*

More and more people are starting to automate their homes, and your home deserves the best tools the industry offers. Ricardo and Mike show you how to use Go, Docker, Prometheus, and Grafana to build top-notch home automation projects you can easily manage with a Raspberry Pi.

→Maik Schmidt

Software Developer

Acknowledgments

To start, we would like to thank the impeccable Jackie Carter for meticulously editing this book. It is because of Jackie's grounding and attention to detail that this book—and all the other Pragmatic books she has edited—has clear explanations and cogent objectives. We can't thank you enough, Jackie, for being there for over a decade!

We would also like to thank our tech reviewers Douglas Clarke, Greg Stewart, John Cairns, Maik Schmidt, Mike Bengtson, and Miki Tebeka, who took time out of their busy schedules and commitments to help refine and polish the final release of this book.

We would also be remiss if we didn't thank the two individuals who brought the Pragmatic Bookshelf to life, providing us and many other talented computing professionals a platform to distribute our enthusiasm, ideas, and passion to our readers. Thank you, Dave and Andy; you guys are the best!

Finally, we would like to thank the creators and maintainers of the Go programming language, the Linux operating systems,

and other open source projects that made this book possible. We appreciate the efforts of the open source community to make such great software available for everyone to use. In particular, we would like to thank Patricio Whittingslow and Scott Feldman, who developed the open source Go Wi-Fi driver for the Raspberry Pi Pico W.

Special thanks from Ricardo:

First of all, I'd like to thank my coauthor, Mike Riley, for the opportunity to work on this great book. Mike is a fantastic human being, a fellow technology enthusiast, and a skillful writer and professional. I am honored to have worked with him on this project and I hope we can work together again in the future.

I'd like to thank my good friend Douglas Clarke for his contributions to this book and for his friendship over the years.

Finally, I'd like to thank my daughters Gisele, Livia, Elena, and Alice for making me the proudest father in the world. Thank you for allowing me the time to work on another book. I love you so much.

Special thanks from Mike:

First and foremost, I have to thank my coauthor, Ricardo Gerardi, who is a powerhouse of inspiration. Being a single parent raising four girls is challenging enough. Couple that with his day job comprised of long hours, shifting technology climates and demanding deadlines—it just makes me shake my head and wonder how he ever found the time to write another book, let alone get any sleep! Ricardo, you seriously rock, man!

Big thanks goes out especially to fellow Pragmatic author Maik Schmidt and my dear friend Mike Bengtson for their outstanding contributions to this book.

Finally, I'd like to thank my family and especially my wife Marinette who has once again sacrificed our time together to allow me to pursue my need to put my consuming joy for technology into words. I love you.

Introduction

Welcome to a new way of thinking about using the Go programming language to automate various facets of your home.

In this book, we walk you through building your Personal Data Center running on a Raspberry Pi. It will use a number of Go-based tools that are commonly employed to monitor large enterprises. Because of Go's remarkable scalability and simplicity, you can install these world-class open source tools that are found in Fortune 500 data centers on a Raspberry Pi, and obtain the same benefits as DevOps engineers and IT professionals across the globe.

Once your management applications are up and running on a Raspberry Pi, we will proceed with building several home automation projects that use the personal data center as a central monitoring and alerting system.

You'll improve your skills by building upon what was learned with each successive chapter, giving you a solid foundation of how to create your own projects afterward. By the end of the

book, you will have the skills to automate nearly anything that uses electrical current in your home, turning dumb appliances into smart ones while using best-in-class Go-based software to monitor, report, and when desired, alert on any activities that might arise during the operation of your solutions.

Who This Book Is For

This book is for developers familiar with the Go programming language who want to do more with it than just the usual integration and microservices that Go is typically used for.

It is also for home automation tinkerers and electronics hobbyists interested in learning how a language like Go can be more powerful and make software projects easier to build and maintain, especially when compared to other languages used in home automation like Perl and Python.

What's in This Book

In Chapter 1, [*Getting Started*](#), you'll review the hardware and software requirements necessary to follow along with building the projects in this book. You'll also learn how to configure some of the software prerequisites and how to select and configure a code editor to write your Go programs.

Next, in Chapter 2, [*Building a REST API Server*](#), you'll use the Go programming language to build a basic REST API service and deploy it on a Raspberry Pi. You'll use this API server later in the book to send notifications from your home automation projects.

In Chapter 3, [*Deploying Your Personal Data Center*](#), you'll deploy your personal data center by assembling and configuring software on a Raspberry Pi that includes the key components for the enterprise-level monitoring and alerting environment. You'll learn how to build containers, capture and report on metrics, and pick up some of our own best practices experiences working with these tools along the way.

Then, in Chapter 4, [*Networking a Temperature Monitor*](#), you'll build your first automation project: a networked temperature monitor that uses a tiny Raspberry Pi Pico W and TinyGo to

gather the ambient temperature and send it to your central monitoring application running on your personal data center.

In Chapter 5, *[Checking the \(Garage\) Door](#)*, you'll build a garage door checker that uses a magnetic switch sensor and the Raspberry Pi Zero 2 GPIO interface to report whether your garage door is open or closed.

Next, in Chapter 6, *[Lighting the Weather](#)*, you'll design a dynamic lighting solution that offers a unique way to visually identify the current outdoor temperature in your area by controlling the colors on a Hue lighting system via APIs.

As the final project, in Chapter 7, *[Watching the Birds](#)*, you'll discover how to build a custom bird feeder that captures high-resolution images of birds—and other wildlife—perched at the feeder, and send those images as attachments to your own designated Discord server channel.

Finally, in Chapter 8, *[Go Build](#)*, you'll review some ideas on how to further improve your skills and how to use the knowledge and experience acquired in this book in other projects.

About the Hardware

Our objective for the book was to avoid as much electrical engineering and wiring as possible. You can complete each project in this book without ever picking up a soldering gun. While it's commendable to use one for appropriate cases, this book focuses more on software than hardware. We also didn't want to have hardware components fail as a result of poor soldering or confusing wiring diagrams, so we opted to make the hardware configuration for these projects as simple as possible to avoid any frustration or expensive mistakes.

As you gain more confidence in your home automation skills using Go, we recommend expanding your horizons with a good basic electronics tutorial and a quality soldering gun. We also recommend continuing to advance your newly acquired skills by experimenting with a variety of electric components found on popular electronics project websites.

About the Code

While it's the goal of this book that you learn something new in Go by working on these featured home automation projects, this book will not cover some of the basics of the language. To follow the examples in this book, we expect you to know how to write basic Go programs that include variables, loops, `if` conditions, and functions. Many resources are available to help you understand the basics of the language. Among those, we suggest:

- A tour of Go:^[1] An interactive guided tour covering Go's main features.
- Powerful Command-Line Applications in Go:^[2] Learn Go by building command line applications.
- Go Brain Teasers:^[3] Explore more advanced language concepts through 25 brain teasers.

Online Resources

You can find more about this book, as well as download the complete project source code online on the Pragmatic Bookshelf website.^[4] You'll find the book forum there, where you can talk with other readers and with us. If you find any mistakes, please report them on the errata page.

We hope you enjoy building the projects as much as we did, and look forward to your comments and photos of your creations along the way. Most of all, have fun bringing life to your ideas using the Go programming language.

FOOTNOTES

[1] <https://go.dev/tour/list>

[2] <https://pragprog.com/titles/rsgo/powerful-command-line-applications-in-go/>

[3] <https://pragprog.com/titles/d-gobrain/go-brain-teasers/>

[4] <https://pragprog.com/titles/gohome>

Part 1

Setup

Before you can start building various home automation projects using Go, you first need to collect the hardware and software resources you'll use to construct the projects. Then we'll create a robust infrastructure to host our projects, using the same scalable tools that large companies and premier technology organizations use in their own IT operational environments.

Chapter 1

Getting Started

Welcome to the fascinating world of home automation. Using home automation, you can improve your home's energy efficiency and security. You can automate repetitive tasks that are often forgotten, like ensuring your garage door is closed at night. You can also control your home remotely, and increase your convenience and comfort.

While there are ready-to-use home automation solutions available in the market, the goal of this book is to teach you how to develop your own solutions. This is a perfect opportunity to practice your Go coding skills while developing something useful, tailored to your specific needs.

In this book, you'll use a small device, the Raspberry Pi, and the Go programming language to develop four home automation projects. These projects are fully functional and fun to build. We've tried to keep the source code examples relatively short; nevertheless, you'll apply different aspects of the Go language that allow you to build robust programs. By working on these

projects, we expect that you'll have some immediate benefits, but we also expect that you'll learn more about the concepts and tools behind them, so you can expand and adapt them to your own needs.

Go^[5] is a fast, statically typed, and compiled language that allows you to build efficient programs with relative ease. Go provides some of the flexibility provided by dynamic languages—such as Python—but the compilation process ensures that you build reliable software that runs efficiently, even on low-powered hardware such as the Raspberry Pi. Go is a versatile language that allows you to build robust software not only for large enterprise projects, but also for small applications, making it a great choice for home automation projects.

Nearly every modern home automation solution requires both hardware and software to make it fully functional. The projects presented in this book are no different. Sensors and controllers need to connect to a computer running code that knows how to interact with those devices. Fortunately, the hardware requirements that are used in this book's projects are inexpensive, and the software we'll use in the book is free, as in open source free. Without the contributions of these dedicated open source contributors, much of what is presented in this book would not be possible. A deep debt of gratitude to all those

developers who have contributed their time and expertise into building great software so we can build great projects.

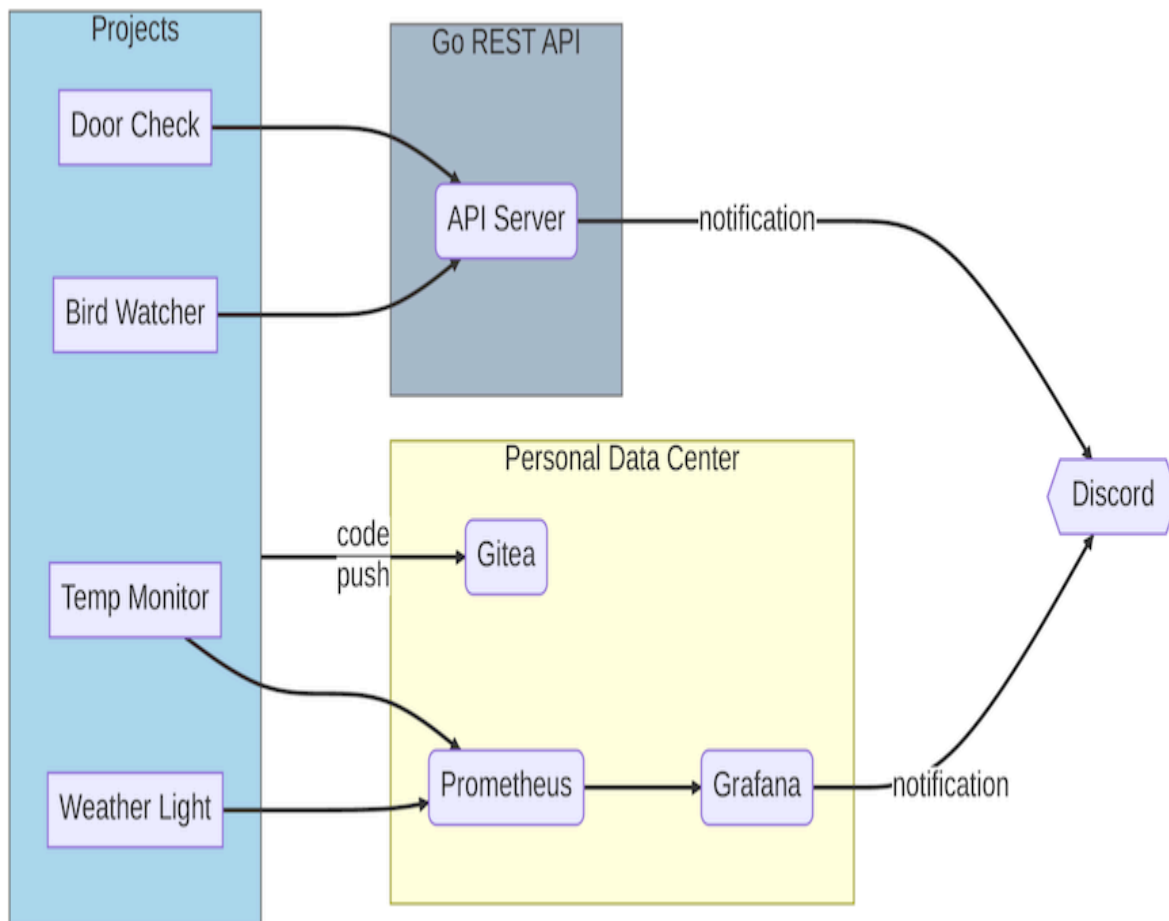
Your Personal Data Center

Before you can start coding the projects, you first need to set up the infrastructure that the Go code will rely upon. In other words, you'll build your own personal data center that will provide the foundation for running many Go-related projects, not just those related to home automation. Your code will rely on these data center components that help send alerts or notifications when thresholds are exceeded, and/or when actions are taken.

Many of the personal data center services of this infrastructure are already written in Go, and are the same ones used in large data centers. These services are ideal because they use industry-standard protocols, granting you a lot of flexibility when building the projects in the book as well as your own future projects. And since these servers and projects are written in Go, your projects will be able to scale to a high degree of activity as an extra benefit of using the language.


Managing your own home automation infrastructure also gives you a deeper understanding of what commercial systems do behind the scenes, including all the potential data collection that may occur without your knowledge. You'll know what the needs of the project are, and what data needs to be captured

and evaluated. If you want to capture and manipulate more details, that's your choice, not the choice of the equipment manufacturer. Take a look at the [diagram](#) to see how these software components and services interoperate to deliver our robust home automation solutions.



This diagram shows how the projects you'll develop in this book make use of the home automation infrastructure you'll deploy in Chapter 2, [Building a REST API Server](#) and Chapter 3, [Deploying Your Personal Data Center](#). Projects "Door Check" and

“Bird Watcher” use the Go REST API to send notifications out, while projects “Temperature Monitor (Temp Monitor)” and “Weather Light” use services from your personal data center to collect and visualize data. Also, you can use Gitea to manage and version control the source code of all your projects in a central location.

 Joe asks:

How did you create this diagram?

The software components diagram was created using Mermaid, [\[6\]](#) an open source JavaScript tool that renders diagrams in different formats from a text-based definition. We chose this approach because it’s easier to create and modify relationship-based diagrams, such as flowcharts and class charts, by defining these relationships as text, without worrying about drawings, positioning, and arrow attachment. Also, because it’s text-based, we can version control the diagrams and keep them in the same repository where we keep our code.

To deploy these services, you’ll need hardware to run it on. In this chapter, we’ll first take a look at the hardware that we’ll use to build the various projects in the book, starting with the Raspberry Pi. After that, we’ll briefly summarize the software and services we rely on for the projects to work effectively.

Once inventoried and set up, you'll build in the next chapters what we like to call your Personal Data Center.

Selecting a Raspberry Pi

The Raspberry Pi has permanently altered the home automation and technology tinkerer landscape by making rather powerful computers and microcontrollers for a fraction of what their counterparts cost. It's still mind-blowing to realize that a Linux server more powerful than rack-mounted servers from a decade ago can fit in the palm of your hand. In the case of the smallest Pi computer, it even takes up less space than a stick of gum.

For a majority of the projects in the book, you'll need, at the very least, a Raspberry Pi 3 Model B+^[7] for roughly \$35. Better still, a Raspberry Pi 4B^[8] for the same price is an ideal tinkering model. It offers enough CPU, RAM, and ports to deliver a flexible platform to develop these and future projects that you may want to create. If you're willing to spare an additional \$20, you can get the latest Raspberry Pi 5^[9] that delivers a two to three times CPU performance increase compared with model 4B.

Another Pi model that will support these projects, and may be beneficial when space constraints are a concern, is the Pi Zero 2 W.^[10] This remarkable piece of hardware is nearly equivalent to the performance of a Pi 3 for a third of the price. However, because it does not have onboard USB, HDMI, Ethernet ports, or

header pins, configuring a Pi Zero takes a bit more effort and additional hardware dongles to properly set up. Once configured, the Pi Zero 2 W is just as manageable and easy to operate and maintain as a more expensive Pi with more options.

One final Pi model that we'll use is not a computer like the previously mentioned Pi's. The Pi Pico W^[11] is a fairly new \$6 microcontroller that has onboard Wi-Fi. Because it's a microcontroller, it has its own language and is designed for very specific jobs. Unlike a multi-purpose computer, the Pico has limited resources and is best suited for targeted monitoring of defined events. We'll use this inexpensive, wireless microcontroller to transmit monitored values to a REST server that will perform additional processing and make cool things happen. You need this microcontroller to work on the project in Chapter 4, *Networking a Temperature Monitor*.

Adding Other Hardware Components

While the Pi provides a powerful, cost-effective foundation to build our projects on, its sensing and controller capabilities only go so far before they need additional hardware to do interesting things. As such, here are additional hardware components that we'll use in the projects featured in this book.

Hue Starter Kit^[12]

Includes the Hue Bridge and two bulbs. The Hue Bridge provides the gateway to communicate with Hue-branded accessories. This includes not only lighting products, but also power switches that can be turned on and off with simple commands. The white bulbs can be turned on and off with variable degrees of brightness, and multi-colored bulbs or light strips can use assigned colors to indicate various degrees of moods or alert states.

Hue Lightstrip Plus Base Kit^[13]

The multi-colored lighting accessory we will use in the Weather Forecast project to visually represent the current outdoor temperature, as well as pulsate when defined temperature limits are exceeded.

Passive Infrared Sensor^[14]

Used to detect motion. We'll use this component in the Bird Watcher project to help detect when a bird (or other critters) approaches a bird feeder.

Magnetic Contact Switch^[15]

Uses a switch and a magnet to indicate if two parts are in contact; for example, indicating when a door opens or closes.

Raspberry Pi Camera Module 2^[16]

A decent HD-quality camera attachment for the Raspberry Pi that we can use to capture both still photos and video. We'll use it to capture pictures of birds when they come to perch and feed at the Pi-powered bird feeder we'll build in the Bird Watcher project.

Female-to-Female Jumper Wires^[17]

These wires will be used to connect the sensors to the Pi header pins.

Optional-Solderless Headers for Pi Zero W^[18]

Those who prefer a solderless approach to attaching header pins to the Pi Zero W. This solution only requires a hammer and a few whacks to securely seat the Pi header pins into place. The pins will make attaching wires to the Pi Zero much easier.

The nice thing about these hardware add-ons is that you can use them not only to extend the projects we'll build, but you can also repurpose them for new project ideas that better suit your specific needs once you get accustomed to building automation solutions with Go.

Now let's look at the software.

Configuring the Software

In addition to the hardware required to complete these projects, you also need some software. You'll use the software to power your Raspberry Pi devices, codify the logic for your home automation projects, develop supporting systems—such as APIs—and run your applications as Linux containers for better isolation. All the software you'll use in the book is open source and available to download from the Internet.

Raspberry Pi OS

To start, you need an operating system for your Raspberry Pi devices. While many options are available out there, some better suited for more advanced users and some beginner friendly, we'll standardize on the official Raspberry Pi OS for the book's projects. This operating system provides a good mix of features and stability, which should make both beginner and advanced users feel comfortable.

Raspberry Pi OS^[19] is available in two flavors: 32 or 64 bit. For the book, we're standardizing the examples using the 64-bit version. If you're running a Raspberry Pi 5, Pi 4, or Pi Zero 2, we recommend using the 64-bit OS version. For Raspberry Pi 3, you have to use the 32-bit version. In this case, all the examples

should work in a similar way but you need to change the target OS from `arm64` to `arm` when building your Go applications.

You can find instructions on how to set up your Pi in the Getting Started with Raspberry Pi [\[20\]](#) webpage.

Go Programming Language

The projects in this book work with any version of Go 1.22 or higher. At the time of writing this book, Go 1.22 is generally available.

Go supports cross compilation, allowing you to build an executable version of your program using a different platform from the one running Go. So, you don't necessarily need to run Go on your Raspberry Pi to complete these projects.

Nevertheless, we wanted to provide the option of using Go in the Raspberry Pi OS for the readers using that platform exclusively.

You'll learn more about using Go on a different platform throughout the book's projects.

If you want to install and use a different Go version than the one shipped with your device, or if you're looking to install Go

in a different operating system, take a look at the Go Downloads^[21] webpage.

TinyGo

TinyGo^[22] is an alternative Go compiler for the Raspberry Pico and Pico W microcontrollers. It is based on the LLVM^[23] toolchain that allows you to compile Go code to run on microcontrollers like the Pi Pico. TinyGo compiles your Go program with special options and optimizations, enabling it to run on limited CPU power and RAM available with such small devices. It also provides a package to interface with device-specific hardware such as IO pins.

For more information and installation instructions, consult the project's Quick Install Guide.^[24]

Docker Community Edition

Docker^[25] is an open source container engine solution that allows you to package and run applications in a self-contained bundle that includes all required dependencies. Packaging your application in a Linux container provides a reliable and repeatable way to run applications on your Raspberry Pi. In addition to including all dependencies, containers isolate applications from each other, allowing you to run applications

that would otherwise conflict with each other, for example, applications that require different versions of the same library or network ports.

To follow the projects featured in this book, you need Docker CE version 24.0 or higher. The version of Docker available in the standard package repository of your Raspberry Pi OS is a bit older, so we recommend installing the latest version using the official Docker repository.^[26] First, add Docker's GPG key used to verify the software signature:

```
$ sudo apt update
$ sudo apt install ca-certificates curl gnupg
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL
https://download.docker.com/linux/debian/gpg | \
    sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Then, add the Docker repository to `apt` source list:

```
$ echo \
    "deb [arch=$(dpkg --print-architecture) \
        signed-by=/etc/apt/keyrings/docker.gpg] \
```

```
https://download.docker.com/linux/debian \
$(. /etc/os-release && echo "$VERSION_CODENAME")
stable" | \
sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
$ sudo apt update
```

Finally, install Docker and its dependencies using `apt`:

```
$ sudo apt install docker-ce docker-ce-cli
containerd.io \
docker-buildx-plugin docker-compose-plugin
```

The installation process starts the Docker daemon automatically. You can verify that it's running with the command `systemctl`:

```
$ sudo systemctl status docker
```

Then, add your current user to the `docker` group to run `docker` commands without `sudo`:

```
$ sudo usermod -aG docker ${USER}
```

Finally, log out, then log back in and run `docker version` to ensure Docker is running and your user can run commands:

```
$ docker version
```

```
Client: Docker Engine - Community
```

```
Version: 24.0.7
```

```
...
```

```
Server: Docker Engine - Community
```

```
Engine:
```

```
Version: 24.0.7
```

If you're using a 32-bit version of Raspberry Pi OS, then follow the instructions available in Docker's documentation.^[27]

Docker Compose

Docker Compose^[28] is a plugin for Docker that provides a relatively easy way to run multi-container applications in a single environment. For example, you can use Compose to run a blog application that depends on two containers, a web server and a database. Compose defines the required containers, storage, and network on a single `yaml` file, and manages the application life cycle by issuing the necessary Docker commands to start and stop the application containers.

The Docker CE package you installed already includes Docker Compose as a standard plugin, but the version of Docker available on Raspberry Pi OS repository does not. The examples in this book use version 2 of Docker Compose included in Docker CE. Make sure you're running this version:

```
$ docker compose version
```

```
Docker Compose version v2.21.0
```

In addition to the software, you also need a way to type your Go programs.

Picking a Code Editor

If you're already familiar with or using a text editor for other purposes, you could use it to program Go as well, since Go source files are regular plain text files. However, some text editors integrate well with the Go tooling, providing a better code experience with support for syntax checking and code auto-suggestion. We recommend one of these two options:

- Neovim^[29] for a lightweight, terminal-based experience that runs well on any hardware, including a headless Raspberry Pi.
- Visual Studio Code^[30] (or one of its open source alternatives) for a graphical experience with many out-of-the-box features.

Neovim comes with basic syntax highlighting available for Go, but to enable more advanced features, you need to install a plugin. We recommend `go.nvim`^[31] which provides full integration with the Go tooling, and a comprehensive set of features for a complete development experience on your terminal. `Go.nvim` requires some additional plugin dependencies, as well as Go tools. For more details, consult the plugin installation instructions.^[32]

When the plugin is installed, install the Treesitter parser for Go, by running the command `:TSInstallSync go` in Neovim:

```
Downloading tree-sitter-go...
Creating temporary directory
Extracting tree-sitter-go...
Compiling...
Treesitter parser for go has been installed
```

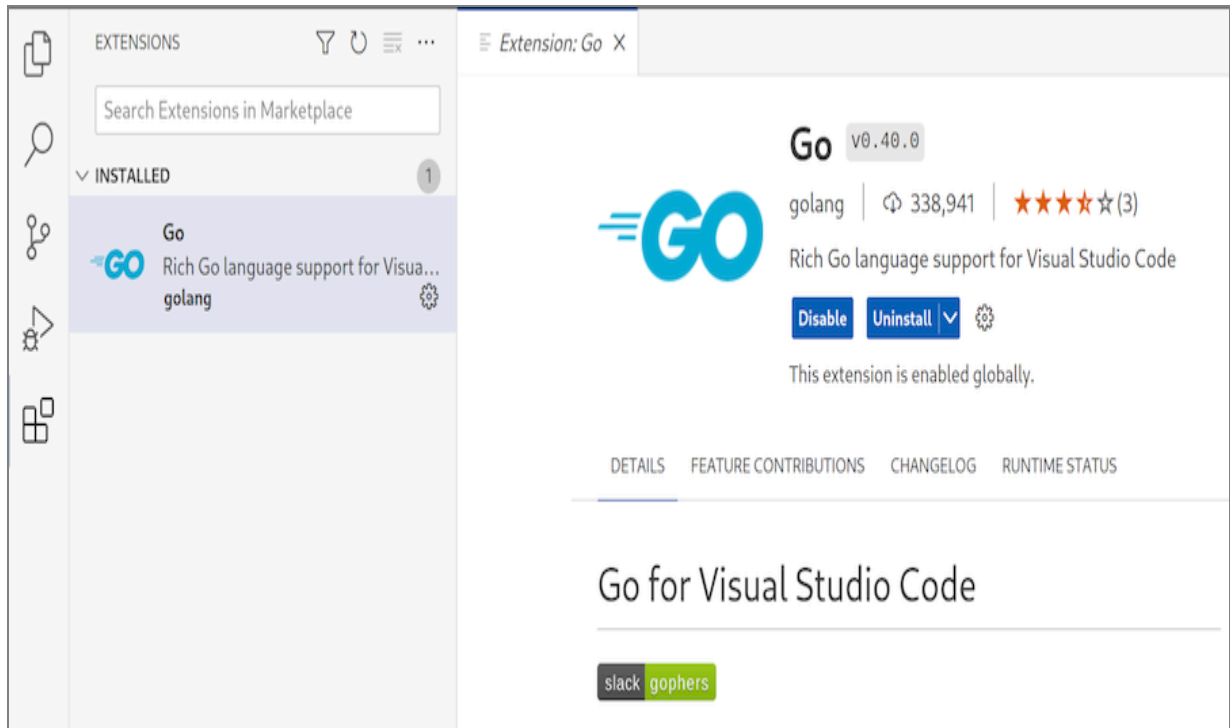
Then, ensure you have all the required Go tooling by executing the command `:GoUpdateBinaries` in Neovim and waiting for the install to complete:

```
install github.com/fatih/gomodifytags@latest finished
install github.com/abenzl267/gomvp@latest finished
install github.com/kyoh86/richgo@latest finished
install github.com/josharian/impl@latest finished
install mvdan.cc/gofumpt@latest finished
install github.com/tmc/json-to-struct@latest finished
install github.com/koron/iferr@latest finished
install github.com/segmentio/golines@latest finished
install github.com/golang/mock/mockgen@latest
finished
install
github.com/davidrjenni/reftools/cmd/fillswitch@late
```

```
finished
install github.com/cweill/gotests/...@latest finished
install
github.com/davidrjenni/reftools/cmd/fillstruct@latest
finished
install github.com/abice/go-enum@latest finished
install golang.org/x/vuln/cmd/govulncheck@latest
finished
install github.com/onsi/ginkgo/v2/ginkgo@latest
finished
install gotest.tools/gotestsum@latest finished
install github.com/go-delve/delve/cmd/dlv@latest
finished
install github.com/golangci/golangci-
lint/cmd/golangci-lint@latest finished
install golang.org/x/tools/gopls@latest finished
install golang.org/x/tools/cmd/callgraph@latest
finished
install golang.org/x/tools/cmd/guru@latest finished
install golang.org/x/tools/cmd/gorename@latest
finished
install golang.org/x/tools/cmd/goimports@latest
finished
```

Now you can use Neovim to develop your Go programs.

To enable Go syntax highlight and code auto-completion in Visual Studio Code, install the official VSCode Go extension, ^[33] as shown in the following image, and restart VSCode.



Once the extension is installed, install the required Go tooling from VSCode by pressing **Ctrl+Shift+p** to open the Command Pallet and run the command **Go: Install/Update Tools**. Select all suggested tools and wait for the install to complete. You can see the following command output in the **OUTPUT** window:

```
Tools environment: GOPATH=/home/ricardo/go
```


Installing 7 tools at /home/ricardo/go/bin in module mode.

- gotests
- gomodifytags
- impl
- goplay
- dlv
- staticcheck
- gopls

Installing

github.com/cweill/gotests/gotests@latest

SUCCEEDED

Installing github.com/fatih/gomodifytags@latest

SUCCEEDED

Installing github.com/josharian/impl@latest

SUCCEEDED

Installing

github.com/haya14busa/goplay/cmd/goplay@latest

SUCCEEDED

Installing github.com/go-

delve/delve/cmd/dlv@latest SUCCEEDED

Installing

honnef.co/go/tools/cmd/staticcheck@latest

```
SUCCEEDED
```

```
Installing golang.org/x/tools/gopls@latest
```

```
SUCCEEDED
```

```
All tools successfully installed. You are ready to  
Go. :)
```

Once all Go tools are installed, you're ready to use VSCode to develop Go programs.

Now that you have all the prerequisites in place, and a code editor configured, you're ready to start developing home automation projects in Go.

Next Steps

In this chapter, we reviewed all the hardware and software required to work on the projects featured in this book. You selected a Raspberry Pi device and analyzed all hardware components required by the different projects. You learned about the required Raspberry Pi operating system and Go versions. Finally, you also installed Docker and Docker Compose on your Raspberry Pi, and configured a code editor to develop Go code.

In Chapter 3, *[Deploying Your Personal Data Center](#)*, you'll set up additional software infrastructure to support your projects. These additional pieces of software enable you to manage source code, perform monitoring and alerting, and visualize data.

But first, in the next chapter, we'll use Go to build a REST API server to pass messages and states between the different components in your home automation projects.

FOOTNOTES

[5] <https://go.dev/>

[6] <https://mermaid.js.org>

[7] <https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/>

[8] <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

[9] <https://www.raspberrypi.com/products/raspberry-pi-5/>

[10] <https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/>

[11] <https://www.raspberrypi.com/products/raspberry-pi-pico/?variant=raspberry-pi-pico-w>

[12] <https://www.philips-hue.com/en-us/p/hue-white-starter-kit-e26/046677563080>

[13] <https://www.philips-hue.com/en-us/p/hue-white-and-color-ambiance-lightstrip-plus-base-v4-80-inch/046677555337#overview>

[14] <https://chicagodist.com/products/adjustable-infrared-pir-motion-sensor>

[15] <https://www.adafruit.com/product/375>

[16] <https://www.raspberrypi.com/products/camera-module-v2/>

[17] <https://chicagodist.com/products/female-female-jumper-wires>

[18] <https://chicagodist.com/collections/pimoroni/products/gpio-hammer-header-solderless-male-female-installation-jig>

[19] <https://www.raspberrypi.com/software/>

[20] <https://www.raspberrypi.com/documentation/computers/getting-started.html>

[21] <https://go.dev/dl/>

[22] <https://tinygo.org/>

[23]<https://llvm.org/>

[24]<https://tinygo.org/getting-started/install/>

[25]<https://www.docker.com/>

[26]<https://docs.docker.com/engine/install/debian/>

[27]<https://docs.docker.com/engine/install/raspberry-pi-os/#install-using-the-repository>

[28]<https://docs.docker.com/compose/>

[29]<https://neovim.io/>

[30]<https://code.visualstudio.com/>

[31]<https://github.com/ray-x/go.nvim>

[32]<https://github.com/ray-x/go.nvim#installation>

[33]<https://marketplace.visualstudio.com/items?itemName=golang.go>

Chapter 2

Building a REST API Server

A primary technology standard that we'll call upon in several of the projects featured in this book is Representational State Transfer, better known as REST. Rather than re-inventing the wheel when it comes to passing state from one machine to another, we can use a robust mechanism that works just as effectively in small projects as it will in top-tier enterprise applications.

REST packages information into JavaScript Notation, or JSON, format and transfers those details using standard HTTP protocol. It's one of the most prevalent ways to transfer meaningful information from one machine to another and it's popular with both consumer and enterprise web applications. In this chapter, we'll create a simple REST server that will provide the foundation for our Raspberry Pi to send and receive state changes. These can range from transmitting the values of a sensor to receiving the results of an executed command. This detail of communication will allow us to build sophisticated and robust workflows that can be monitored and

triggered from a variety of REST-aware tools. We'll use some of these tools to test the REST server once we have it up and running.

After writing the code to execute the server, we will package the server into an application container. Doing so allows for easy deployment and robust uptime, since the container will automatically restart the server in the event of a crash or reboot. These will be the same steps we will use to deploy the applications throughout the book into our Raspberry Pi server environment.

Writing the Code

While we could use any of the popular web application frameworks created by the Go community, such as Gin^[34] or Fiber,^[35] Go's built-in libraries have all the functions we need to build a lightweight REST API server. Taking this approach greatly reduces the number of external library dependencies required to run our server and significantly reduces the system resources required to run the server. And for those security-conscious readers, this design decision also reduces the potential attack surface by limiting the number of potentially exposed unnecessary features that will be compiled into the deployable executable. This bulk reduction will come in especially handy when we package and execute the server within an application container.

Start a new Go project by creating a directory called `restapi` and issuing the following Go command within that directory:

```
$ go mod init gohome/restapi
```

Next, using your preferred editor, create a new file within the `restapi` directory called `main.go` and begin by importing the built-in packages we'll use to create the REST API server.

```
package main
```



```
import (  
    "encoding/json"  
    "fmt"  
    "net/http"  
    "os"  
    "os/exec"  
)
```

The JSON encoding library will be used to process inbound data and format outbound JSON outputs. The `fmt` standard library formats string output. The HTTP library is used to manage our HTTP server connection. The `os/exec` library allows Go to call other applications installed on the machine.

Now create a custom Go type called *cmdresult* using a `struct` keyword to instruct Go how to format the JSON being emitted from a REST endpoint.

```
type cmdresult struct {  
    Success bool    `json:"success"`  
    Message string `json:"message"`  
}
```

This JSON structure contains two attributes, **Success** and **Message**, indicating whether or not the function call was successful and any specific message to be transmitted as part of that success result.

Next up, create a function to respond when the root level of the server path is called from a browser. While not essential, having this kind of function offers a quick sanity check to see if the server is up and running.

```
func homepage(w http.ResponseWriter, _
*http.Request) {
    fmt.Fprintf(w, "Go Home Simple REST API
Server")
}
```

Next, create the primary function to handle an inbound request that will query and return the server's current date, transmitted in our defined JSON-formatted response. This is an example of how to use a system command to obtain information you can return in a RESTful way. Later in the book, you can use a similar approach to run other commands, such as the command to activate the Raspberry Pi camera.

```

func getdate(write http.ResponseWriter, _
*http.Request) {
    result := cmdresult{}

    out, err := exec.Command("date").Output()
    if err == nil {
        result.Success = true
        result.Message = "The date is " + string
(out)
    }

    json.NewEncoder(write).Encode(result)
}

```

Declare the **result** variable and initialize it to an empty **cmdresult** struct. Then, using the **exec.Command** function from the *os/exec* package, call the external **date** program and capture its output. If this call succeeds, set the **Success** boolean attribute value of **result** to **true** and the **Message** to a sentence containing the command output. Otherwise, these attributes remain with the **zero** values of **false** and empty string. Then encode the **result** struct to JSON and return the result via HTTP to the caller.

Lastly, initialize the HTTP server, set the port number and declare the relative paths that will be used to call the respective

homepage and getdate functions.

```
func main() {  
    http.HandleFunc("/", homepage)  
    http.HandleFunc("/api/v1/getdate", getdate)  
    err := http.ListenAndServe(":4000", nil)  
    if err != nil {  
        fmt.Println("Failed to start server:",  
err)  
        os.Exit(1)  
    }  
}
```

Save this `main.go` file and run the server via the `go run` command.

```
$ go run main.go
```

If there are no syntax errors, the server should be running and listening on port 4000 for an inbound connection. Open your preferred browser and type in the IP address of the machine you are running this REST API server on, followed by the port number. For example, if you are running the browser on the same host that is running the server, enter <http://localhost:4000>. If the server is running and functioning correctly, you should see “Go Home Simple REST API Server” in your browser window.

Verify that the `getdate` functionality is working correctly by entering its relative path into the browser, such as <http://localhost:4000/api/v1/getdate>. If successful, the browser window should display the success state and the date string in JSON-formatted syntax.

```
{ "success":true, "message": "The date is Sat Dec 2  
10:25:31 AM EST 2023" }
```

Congratulations! You have built the first building block of an important service that will allow your servers to communicate with one another in a standard fashion. Now let's package it into an application container, trimming it down to its most lightweight core essentials.

Containing the Server

Now that your API server is ready, you can run it on your Raspberry Pi. Because Go is a compiled language, you could compile and run it directly on the operating system. However, to increase the deployment flexibility, let's package the API server into an application container. A container allows you to package an application with all of its dependencies, and to run them in isolation from other system processes. By doing this, you can run multiple instances of your application, or other applications on the same machine, even if they have conflicting dependencies or requirements, such as TCP ports.

Running the application as a container also allows you to take advantage of some features the Docker container engine provides. One such feature automatically restarts the container in case of failures or system restart. This ensures that your Raspberry Pi Linux operating system keeps the contained server always up and running automatically.

Before you can package your application in a container, you need to compile it. Let's do that next.

Compiling Your API Server

Running your Go application by using the command `go run` is a nice and quick way to verify if your application is working. It's particularly useful during development time, but to run it in a production environment, you need to compile your application into a binary file.

In Go, the compilation process is also known as “building” and you run it by using the command `go build`. In its most basic form, running `go build` builds your application into a single binary file compatible with your current operating system and architecture. It also dynamically links your application to the current system libraries. This is usually enough to run the application on your machine. However, using this approach isn't the best way to run a Go application inside a container.

When building your Go application, you can specify additional environment variables or parameters to customize the build and optimize the resulting binary. Let's look at some common environment variables:

- `GOOS`: Defines the target operating system for the build. Defaults to the current operating system. Go allows cross compilation, which means you can compile your Go application for a different operating system from your

current system. To build the API server binary to run in a Linux container, set this variable to “linux”.

- **GOARCH**: Defines the target processor architecture for the build. Defaults to the CPU architecture of the system running the build. To build your API server binary to run on Raspberry Pi, set this variable to “arm64”.
- **CGO_ENABLED**: Defines whether or not the resulting binary dynamically links to the system libraries. To build a static binary to run in a container, set it to “0” (zero).

You can see all possible combinations of operating system and architecture to use with **GOOS** and **GOARCH** by running `go tool dist list`.

In addition to variables, you can also pass additional parameters to `go build`. To optimize your binary to run in a container and decrease its size, pass the link parameter `-ldflags` set to `-s -w` to strip the resulting binary of all debugging symbols. You can learn more about additional build options by running `go help build`.

Build your API server binary with all these options by running `go build` like this:

```
$ CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build -ldflags="-s -w"
```


This command creates a binary file, `restapi`. Use the Linux command `file` on it, to verify build details:

```
$ file restapi
restapi: ELF 64-bit LSB executable, ARM aarch64,
version 1 (SYSV),
    statically linked, Go BuildID=QSMA....CAxpj,
stripped
```

Next, let's create a container image to allow your API server to run as a container.

Creating a Container Image for Your Server

To run your API server as a container, you must first create a container image for it. A container image packages your server application with required dependencies using a standard format, supported by different container engines such as Docker or Podman.

If you haven't done so, follow the instructions on [Docker Community Edition](#), to install Docker on your Raspberry Pi.

Then, specify the instructions on how to build the image in a configuration file named `Dockerfile`. This file works as a recipe with all the steps needed to create a container image.

Since you have your API server binary pre-compiled, you could use it to create the container image. However, to make the process reproducible, let's create a multi-stage Dockerfile, where the first stage uses the official Go image to build the server binary, and the second stage generates the final image to run your server. By applying this technique, you can rerun the container build process to build the binary and create the image when you modify or update your code, avoiding a manual recompilation. Define your multi-stage Dockerfile like this:

```
FROM docker.io/golang:1.22 AS builder
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build
  -ldflags="-s -w"

FROM docker.io/alpine:latest
RUN mkdir /app && adduser -h /app -D restapi
WORKDIR /app
COPY --chown=restapi --from=builder /app/restapi .
EXPOSE 4000
CMD ["/app/restapi"]
```

Another advantage of running a multi-stage build is that it allows you to use a Go base image to build the application with a different version from the one installed on your system. For example, when we were writing this book, Raspberry Pi OS was packaging Go version 1.19, while we specified a more recent version of Go, 1.22, in the Dockerfile used to build the server for the container.

Save this file with the name `Dockerfile` and build your server image by using the `docker build` command. Tag your image with the identifier “restapi:v1” using option `-t`. Use a period `.` at the end to define the current directory with the Dockerfile as the build context:

```
$ docker build -t restapi:v1 .
```

This command builds the image by following the steps defined in Dockerfile, downloading any required images, building the server, and finally creating the API server image by copying the server binary in a base Alpine Linux image. When the command finishes building the image, you can see it by listing locally known images using the command `docker images`:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
restapi	v1	a00767dd3fa8	About a
minute ago	12.2MB		
golang	1.22	4c88d2e04e7d	11 days ago
851MB			
alpine	latest	5053b247d78b	5 weeks ago
7.66MB			

Test your image by running your REST API server container with `docker run`. Use option `-d` to run it as a background service, and `-p 4000:4000` to expose port 4000 for external connection:

```
$ docker run -d -p 4000:4000 --name restapi-v1 restapi:latest
57c165d15840fb7bcf79989a1faf0cc1e9fcaf1ac7da823b62e
```

Verify your container is running with `docker ps`:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f68e611b5be7	restapi:v1	"/app/restapi"	37 seconds ago	Up 35 seconds		

```
0.0.0.0:4000->4000/tcp  restapi-v1
```

Finally, verify that it works by submitting a request using `curl`:

```
$ curl http://localhost:4000/api/v1/getdate
{"success":true,"message":"The date is Sat Dec  2
14:29:23 UTC 2023\n"}
```

Now that you have a working container image, let's ensure the container is always running.

Running Your Server Container as a Service

The final step in deploying this container is to have it automatically start up when your Raspberry Pi powers on. This ensures that the server is always up, since the service will automatically restart the server if it stops or crashes. You can do this in different ways, such as creating a script or running helper programs such as `supervisor`. These alternatives require additional maintenance. Docker provides the parameter `--restart` for the `docker run` commands that allows you to control the container restart behavior. Docker documentation does not recommend using a process manager such as Systemd to manage containers.^[36] For this book's projects, we'll use the recommended approach by specifying the parameter `--`

`restart=always` to always restart the container in case of failures or system restarts.

First, stop and delete the running `restapi` container if it's running:

```
$ docker kill restapi-v1 && docker rm restapi-v1
restapi-v1
restapi-v1
```

Then, rerun the container specifying the parameter `--`

`restart=always`:

```
$ docker run -d -p 4000:4000 --name restapi-v1 --restart=always
restapi-v1
ddcb3253ca5218c9e00e13fd9bfc4d2cf8816bf216eb686dab0
```

Verify the correct restart policy is set by using `docker inspect`:

```
$ docker inspect restapi-v1 --format="{{.HostConfig.RestartPolicy}}"
{always 0}
```

You can also restart your Raspberry Pi to ensure the service comes up automatically when the system starts.

If you're using the Podman container engine, you can also create a Systemd unit file using `podman generate`.

Congratulations, you have a running API server that restarts automatically in case of issues and starts automatically every time you start your Raspberry Pi.

Next Steps

With a basic REST server up and running, you now have a way to receive and send messages to our base Pi server in a reliable, standardized way. You've seen how to compile this Go server and package it into a container for Docker to manage. We also used Docker restart options to ensure this container will automatically start whenever the Raspberry Pi hosting it boots up.

With this foundation, we'll build the other projects in this book using a similar approach. After the Go code for a project has been compiled and the resulting executable packaged into a container, we'll deploy and run the project in the same way we did for this REST server.

In the next chapter, you'll learn more about Go-based tools that make application service monitoring and visualization easier. This will culminate in a condensed version of a data center that fits within the confines of a Raspberry Pi's system resources while still giving us plenty of room to run our custom projects. See you there!

FOOTNOTES

[34]<https://gin-gonic.com>

[35]<https://gofiber.io>

[36]<https://docs.docker.com/config/containers/start-containers-automatically/>

Chapter 3

Deploying Your Personal Data Center

Now that you have our application requirements in place and deployed our Go REST API server, you can complete the environment setup by spinning up a few more Go-based applications. These applications will help monitor running services, visualize these results, and send alerts whenever these metrics exceed defined thresholds that need attention.

Essentially, you'll build a condensed version of a data center that fits within the confines of the Raspberry Pi's system resources while still giving you plenty of room to run your custom projects.

You'll accomplish this by deploying three additional pieces of software onto the Pi: Gitea, Prometheus, and Grafana. These open source Go-coded tools allow you to manage source code for your projects, collect metrics and alerts on them, and display them in a nice dashboard. While typically used by DevOps personnel in enterprise settings, the combination of

Go's leanness and the containers' encapsulated deployments make these three applications available for you to run at home, even on low powered hardware like the Raspberry Pi.

The goal of this chapter is to deploy all three of these applications onto a single Raspberry Pi, allowing you to have all the IT infrastructure required for the book's projects in a single box. To ensure these applications do not conflict with each other, as well as with other applications you may be running on your Pi, we'll encapsulate and deploy them as Linux containers.

Deploying Applications with Linux Containers

The concept of running applications in isolation from other workloads in a single shared system is not new, and has existed in different operating systems for many years. Solutions to address this requirement range from simply running a process in a rooted environment—changing the root of the process’s file system—to more advanced and fully isolated virtualized or system partitioned environments. From the Linux kernel perspective, another application isolation method has gained traction in the past ten years: Linux Containers.

Linux Containers allow you to run your applications in isolation from other processes running on your system by applying two important Linux kernel features:

Namespaces

Allows a process to see a subset of system resources as the entire system.

Cgroups

Permits the system to limit and assign hardware resources, such as CPU and memory, to individual processes.

In conjunction with other tooling and container management solutions, such as Docker or Podman, you can use containers to encapsulate an application with all its required dependencies, and run it in isolation from other applications in your system. Running applications in containers prevents conflicts, such as dependency clashes, and allows fine-grained assignment of system resources to each individual application. Particularly, by using containers, you can deploy all the infrastructure required for the book's projects in a single Raspberry Pi, which could be challenging otherwise.

To make it easier to deploy containerized applications in Linux, you can use a container management solution, such as Docker or Podman. These solutions are not required, but they make it easier to package applications into images, as well as manage their life cycle by starting, stopping, managing logs, mounting external volumes, and integrating networking into the containers. As discussed in [*Docker Community Edition*](#), the examples in this book use Docker as the container management solution. If you haven't done so yet, it's a good idea to follow the instructions in that section to install Docker on your Raspberry Pi.

You can deploy all the services featured in this chapter by using `docker` in a similar way to how you executed your API server in

[Containing the Server](#). The main difference consists in using pre-defined container images provided by the application developers, instead of creating your own images.

In the end, you can also run these containers as services, by following the same procedure you used to create a service for your Go REST API in [Running Your Server Container as a Service](#). It's recommended that you have this infrastructure always running even after your Pi reboots.

The applications featured in this chapter are slightly more complex than the REST API server you developed in the previous chapter. These applications may have additional dependencies such as a database, or rely on additional networking or local storage to ensure it preserves data across restarts. To manage all aspects of each application, you'll use Docker Compose to configure all required dependencies using a single `yaml` configuration file. Docker Compose also allows managing the entire set of required components with a single command that starts and stops the application components in the correct order. If you haven't done so, follow the instructions in [Docker Compose](#) to install Docker Compose on your Raspberry Pi.

Let's start by deploying a local Git front end, Gitea, to manage source code for your projects, by using Docker Compose.

Managing Source Code with Gitea

Let's start your IT infrastructure in a box by deploying a local Git web front end and server to help you manage your project's source code. While this is not required to complete the projects, it's a good idea to have a private place to manage your source code. Gitea is a popular GitHub clone written in Go that, in addition to managing your source code with Git, provides many of the features available in GitHub such as recording issues, pull requests, and a wiki so you can collaborate with others, as well as maintain documentation together with your source code. It also allows pushing and pulling code using both HTTP and SSH protocols. For more information, consult the project's documentation.^[37]

The Gitea developers provide an official container image you can use to run the server part as a container. In addition to that, Gitea requires a database to store project data. It supports different types of databases such as PostgreSQL, MySQL, and SQLite. For this project, we're using the official PostgreSQL container image to run the database as a container.

To work properly, Gitea requires network connectivity to PostgreSQL database using its standard communication port 5432. To ensure both containers can talk to each other, you can

run them using Docker Compose. By default, when you run multiple containers as a “service,” Docker Compose creates a dedicated Docker Network, allowing all containers to communicate with each other by using their container name as the DNS name. You can also manage other aspects of your containers such as volumes for data persistence, exposed network ports, and environment variables.

To run both containers required for Gitea, create a [docker-compose](#) definition file like this:

[infrabox/gitea/docker-compose.yml](#)

```
version: '3'

services:
  gitea:
    image: docker.io/gitea/gitea:1.20.1
    restart: always
    volumes:
      - "git_data:/data"
    ports:
      - 3000:3000
      - 2222:22
    environment:
      DB_USER: gitea
```

```
DB_NAME: gitea
DB_PASSWD: gitea
DB_TYPE: postgres
DB_HOST: db:5432
db:
  image: docker.io/postgres:15.3-alpine
  restart: always
  volumes:
    - "db_data:/var/lib/postgresql/data"
  expose:
    - 5432
  environment:
    POSTGRES_PASSWORD: gitea
    POSTGRES_USER: gitea
    POSTGRES_DB: gitea
volumes:
  git_data: {}
  db_data: {}
```

In this configuration, notice that you're using environment variables to configure PostgreSQL as well as configuring Gitea to access the database. You're also using Docker volumes in both containers to persist data in case the container restarts, as well as setting the containers to automatically restart in case of

failure or system restarts in the same way you did in [*Running Your Server Container as a Service*](#).

Now, start the service by using the command `docker compose up` in the same directory where you have the configuration file `docker-compose.yml`. Docker Compose automatically looks for a file with this name and uses it for service definition. If your file has a different name, use option `-f` to specify the file name. Use the option `-d` to run the containers in detached mode:

```
$ docker compose up -d
[+] Running 19/19
  ✓ db 8 layers                                0B/0B      Pulled
31.4s
  ✓ 8c6d1654570f Already exists
0.0s
  ✓ d285f268e220 Pull complete
0.4s
  ✓ a4388b4d68d1 Pull complete
0.7s
  ✓ 7f8d1bfa3d3b Pull complete
29.7s
  ✓ 94da97297dcb Pull complete
30.0s
```

```
    ✓ 8262b2843785 Pull complete
30.3s
    ✓ c368c6a56404 Pull complete
30.5s
    ✓ 60f35adbbf32 Pull complete
30.7s
    ✓ gitea 9 layers                                0B/0B      Pulled
31.0s
    ✓ 3920a99c0b41 Pull complete
11.5s
    ✓ ab329f515238 Pull complete
11.8s
    ✓ 65129cc9ce82 Pull complete
12.1s
    ✓ 73011ce1559c Pull complete
23.8s
    ✓ b58a6c8d293b Pull complete
24.6s
    ✓ 87ca5a77c65f Pull complete
24.9s
    ✓ 37a7bbf1d1c1 Pull complete
29.7s
    ✓ a9f0b5401d37 Pull complete
30.0s
```

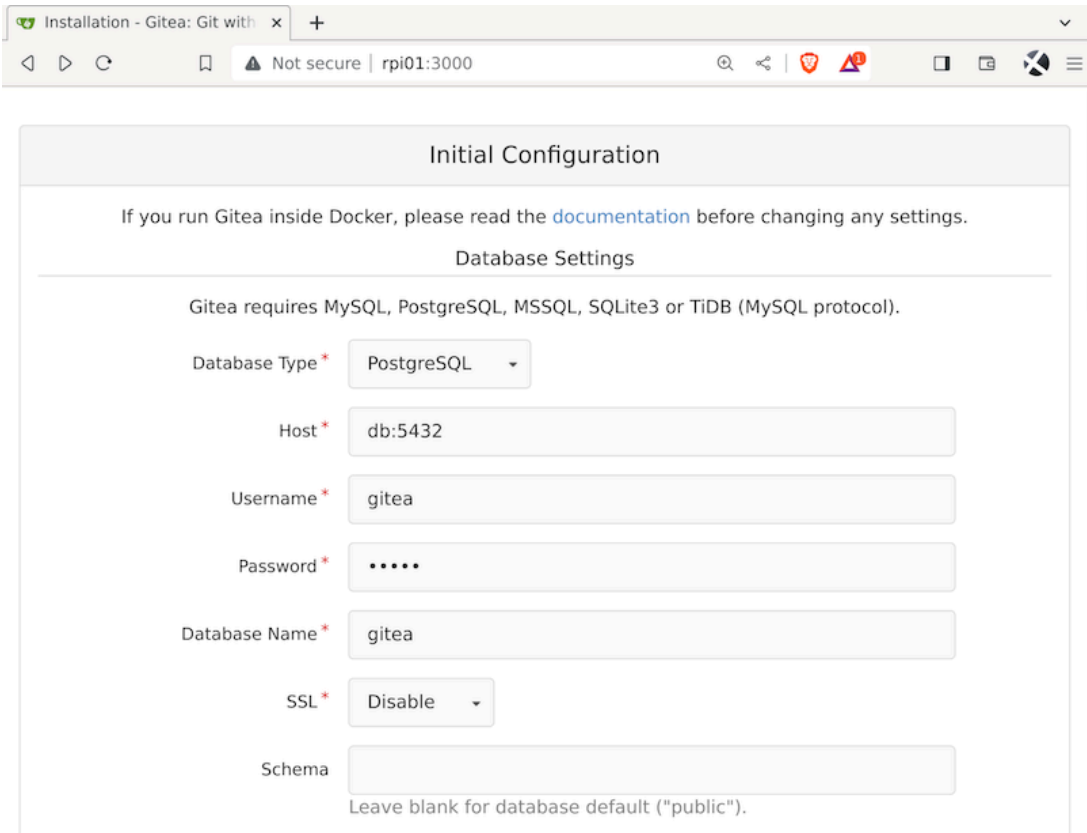
```
    ✓ 85b16303d533 Pull complete
30.2s
[+] Running 4/4
    ✓ Volume "gitea_db_data"    Created
0.0s
    ✓ Volume "gitea_git_data"   Created
0.0s
    ✓ Container gitea-gitea-1    Started
6.2s
    ✓ Container gitea-db-1       Started
6.1s
```

Verify that both Gitea and PostgreSQL containers are running by using the command `docker compose ls`. This command lists the running compose project named `gitea` with two running containers as expected.

```
$ docker compose ls
NAME                STATUS              CONFIG FILES
gitea               running(2)
/home/ricardo/gitea/docker-compose.yaml
```

In the compose configuration file, you also exposed network ports 3000 for Gitea web and 2222 for SSH access. Configure Gitea by accessing its web interface on port 3000. Point your

browser to the Raspberry Pi's IP address (or hostname) on port 3000 to access the initial configuration page, as shown in the [screenshot](#).



The screenshot shows a web browser window with the title "Installation - Gitea: Git with x". The address bar shows "Not secure | rpi01:3000". The page content is titled "Initial Configuration" and includes a warning: "If you run Gitea inside Docker, please read the [documentation](#) before changing any settings." Below this is the "Database Settings" section. It states: "Gitea requires MySQL, PostgreSQL, MSSQL, SQLite3 or TiDB (MySQL protocol)." The form fields are as follows:

- Database Type: PostgreSQL (dropdown menu)
- Host: db:5432
- Username: gitea
- Password: (masked with dots)
- Database Name: gitea
- SSL: Disable (dropdown menu)
- Schema: (empty text box)

Below the Schema field, it says: "Leave blank for database default ('public')." A blue vertical bar is visible on the right side of the page.

Most of the configuration is correctly set from the environment variables you added to the compose configuration file. You need to adjust three properties to ensure your Gitea server works:

- Server Domain: Set to your Pi's hostname or IP address
- SSH Server Port: Set to 2222 to match the exposed container port
- Gitea Base URL: Update [localhost](#) with Pi's hostname or IP address

Make the required configuration as shown in the next screen capture:

Then, scroll to the bottom of the page, expand the “Administrator Account Settings” section, and fill in the fields to create an administrator account to manage your server. When you’re done, click the button “Install Gitea” to complete the install, as indicated in the following screen capture:

When the installation is done, Gitea redirects you to its web interface and you’re ready to use it, as shown in the next image:

You can create your repositories from here and access Gitea using SSH on port 2222.

Next, let’s set up a monitoring and alerting infrastructure using Prometheus.

Monitoring and Alerting with Prometheus

Some of the home automation projects featured in this book require monitoring and alerting capabilities. For example, projects in Chapter 4, *[Networking a Temperature Monitor](#)*, provide data that's best viewed when plotted over time, while Chapter 6, *[Lighting the Weather](#)*, requires alerting capabilities. Instead of developing a custom solution to support these applications only, let's roll out an instance of Prometheus, a popular monitoring and alerting solution, using containers on your Raspberry Pi.

Prometheus^[38] is a monitoring system and time-series database widely used to monitor IT infrastructure, particularly container-based workloads. Prometheus is developed using Go, and its flexibility and scalability allow it to capture a variety of metrics, from small environments to large corporate data centers.

Prometheus works by “scraping”—or polling on a fixed scheduled basis—systems for metrics and storing them in a time-series database. It offers a powerful and fast query language to retrieve and evaluate this data for correlation, visualization, and alerting.

The target-monitored systems provide metrics to Prometheus via “exporters.” Prometheus ships several default exporters to monitor common infrastructure such as operating systems and databases. It also provides libraries for different programming languages to develop custom exporters for your applications. Later, in Chapter 4, [Networking a Temperature Monitor](#), you’ll use Prometheus’s Go libraries to develop a custom exporter for your temperature monitoring application.

For now, let’s deploy the required Prometheus infrastructure to monitor your applications using a container. Since Prometheus is commonly used to monitor container workload, the project provides default container images to run the different required components.

Unlike Gitea, the Prometheus server runs as a standalone container without any dependencies so you don’t need to use Compose to deploy it. However, we also want to deploy a second container running a default [node-exporter](#) to collect and expose data from the Linux OS on your Raspberry Pi host. In this case, let’s use Docker Compose to manage both the Prometheus server and the node exporter containers.

First, create a custom Prometheus configuration file. Most of these configurations are standard, and you can find a sample

file and detailed explanation about this configuration in the project's documentation page.^[39] By default, your Prometheus instance will scrape the target system for metrics every fifteen seconds. Change to a different value if you want it to be more or less aggressive. In the `scrape_configs` section, you have the default configuration to collect metrics from itself as `job_name: prometheus`.

By default, obtaining metrics from additional targets requires adding another job to the configuration file for each new target and restarting Prometheus. To make the task of adding new target systems slightly easier, we're adding a file-based service discover job named `svc_discovery` that automatically imports configuration from files that match the specified rule every thirty seconds.

[infrabox/prometheus/prometheus.yml](https://infrabox.com/prometheus/prometheus.yml)

```
---
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 15s
alerting:
  alertmanagers:
  - follow_redirects: true
    enable_http2: true
```

```
    scheme: http
    timeout: 10s
    api_version: v2
    static_configs:
      - targets: []
scrape_configs:
- job_name: prometheus
  honor_timestamps: true
  scrape_interval: 15s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: http
  follow_redirects: true
  enable_http2: true
  static_configs:
    - targets:
      - localhost:9090
```

```
- job_name: svc_discovery
  file_sd_configs:
    - files:
      - '/prometheus/sd_*.json'
      - '/prometheus/sd_*.yaml'
      - '/prometheus/sd_*.yml'
```

```
refresh_interval: 30s
```

When Prometheus runs with this configuration, you can add a new target system by creating a job file whose name starts with `sd_` and with a file extension `yml` or `json`, under the directory mapped to the Prometheus data directory `/prometheus`.

Prometheus reads these files and automatically adds the job to its queue for scraping, without restarting.

Now, create the `docker-compose.yml` file to configure the Prometheus service with the required containers:

[infrabox/prometheus/docker-compose.yml](#)

```
version: '3'

services:
  prometheus:
    image: quay.io/prometheus/prometheus:v2.45.0
    restart: always
    volumes:
      - "prom_data:/prometheus"
      - "
① ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    ports:
      - 9090:9090
```

```
    networks:
②      - prom_net
    extra_hosts:
③      - "rpi-host:192.168.38.1"
    node:
        image: quay.io/prometheus/node-exporter:v1.6.0
        command:
④      - "--path.rootfs=/host"
⑤    pid: host
        restart: always
        volumes:
            - ":/:/host:ro,rslave"
        expose:
            - 9100
⑥    network_mode: host
    volumes:
        prom_data: {}

networks:
    prom_net:
        driver: bridge
⑦    ipam:
        config:
            - subnet: 192.168.38.0/24
```

```
gateway: 192.168.38.1
```

In this Compose configuration, you're using these options, in addition to what you did in [*Managing Source Code with Gitea*](#):

① Mount the configuration file you previously defined in the container.

Use a custom network 'prom_net' allowing extra

② configurations.

Define an additional host 'rpi-host' pointing to the 'prom_net'

③ gateway which resolves to the host, allowing communication with the node container running on the host network.

Provide additional parameters to the command Docker runs

④ when starting the node container so it can use the host root file system.

Use the host PID namespace so the Node exporter can

⑤ monitor host processes.

The Node Exporter requires running on the host network

⑥ instead of a Bridge network so it can monitor the host.

Configure a pre-defined IP range for the 'prom_net' so

⑦ containers on this net can communicate with each other,

but can also define a fixed host name allowing containers to talk to the host network.

When the container is up and running, point your browser to your Pi's hostname or IP address and port 9090 to see

Prometheus's web interface, as shown in the next screenshot:

Currently, there aren't many metrics to see, as the only target Prometheus is monitoring is itself. You can verify the list of target systems by clicking on "Status->Targets" at the top of the screen, as in the following screenshot:

Even though the Node Exporter container is running, Prometheus is still unaware of it. Let's configure Prometheus to scrape it for metrics. By default, the Node Exporter exposes metrics on port 9100. Since we're using the host network to run this container, you can get the metrics by querying the host interface. In the Prometheus container, we mapped the host to the hostname `rpi-host`. Thanks to the service discovery job you added to the Prometheus configuration, you can automatically configure Prometheus to obtain metrics by creating file `sd_node01.yml`, which specifies the target configuration, then copying this file to the exposed volume `prom_data`. Use the hostname `rpi-host` that you added to Prometheus when starting the container to query the host interface from within the container:

[infrabox/prometheus/sd_node01.yml](#)

```
---  
- labels:
```

```
job: node01

targets:
  - 'rpi-host:9100'
```

Use `docker cp` to copy the file to the required path, which will place it in the exposed volume for persistence:

```
$ docker cp sd_node01.yml prometheus-prometheus-1:/prometheus
```

Wait a few seconds and refresh the Prometheus Targets page to see the new target automatically included, as demonstrated in the [screenshot](#).

Targets

All scrape pools ▾ All Unhealthy Collapse All Filter

Unknown Unhealthy Healthy

prometheus (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	11.834s ago	26.416ms	

svc_discovery (2/2 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://rpi-host:9100/metrics	UP	instance="rpi-host:9100" job="node01"	4.115s ago	122.474ms	

Navigate to “Graph” at the top menu, and type `node` in the search box to see how many node metrics are now available. Select one, for example, `node_load5`, to view the system load, and execute it to obtain the current value. Select the “graph” tab to graph it over time, as show in the next screen capture:

Prometheus is a great tool to collect, process, and store metrics. While it has some graph capabilities built in, it’s not the best tool for data visualization. Let’s improve that by deploying Grafana next.

Visualizing Data with Grafana

Grafana^[40] is an open source data visualization tool. Grafana is developed in Go and it enables the creation of detailed dashboards to display data from several applications, such as databases, and monitoring systems like Prometheus.

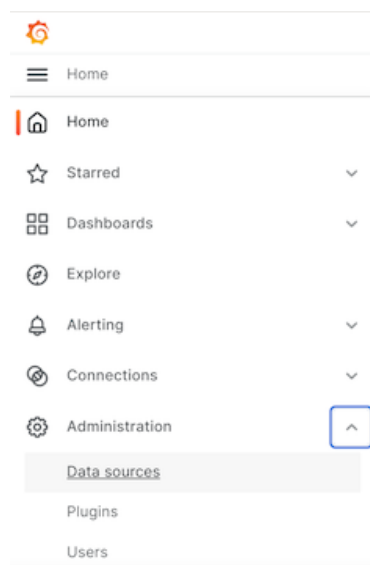
Grafana allows users to install ready-to-go dashboards from JSON files, as well as create their own dashboards in an interactive web user interface. You can combine data from different sources in a single dashboard, navigate and explore metrics, and generate alerts.

Run Grafana as a container on your Raspberry Pi using the same network `prometheus_prom_net` you used for the Prometheus container in [*Monitoring and Alerting with Prometheus*](#), which will make it easier to query and obtain data from Prometheus later. Use option `-p 3100:3000` to map port 3100 on the host to Grafana standard port 3000 in the container. This is required since your Gitea container already uses host port 3000 and they would conflict.

```
$ docker run -d \  
    --name=grafana01 \  
    --restart=always \  
    grafana/grafana
```

```
--net=prometheus_prom_net \  
-p 3100:3000 \  
docker.io/grafana/grafana-oss:9.5.6
```

Now point your browser to the Pi's hostname or IP address on port 3100 to access Grafana's initial screen. Log in with the default user **admin** and password **admin**. On your first login you're required to update the admin's password to proceed. After you're logged in, add a new data source to obtain data from Prometheus, by navigating to "Administration -> Data sources" on the left menu, as indicated in the [screenshot](#).



Then click "Add data source" and select "Prometheus" from the list of available data sources to add. Configure the target **URL** as **<http://prometheus:9090>** to query the Prometheus instance running on the local Pi via the **prom_net** network. Click "Save & Test" to save

and test the connection. You should see a message saying **Data source is working**, confirming the connection is successful.

From this point, you could start querying Prometheus's metrics and build your own dashboard. To get a head start, you can also use an open source dashboard available on the Internet. Many options are available out there but, as an example, let's use an open source dashboard Node Exporter Full,^[41] which provides a comprehensive overview of your operating system's metrics.

To import this dashboard into Grafana, first download the source JSON file from the project's page.^[42] Then, in Grafana, navigate to "Dashboards" and select "Import" from the "New" dropdown menu, as shown in the next screenshot:

Then, upload the dashboard JSON file, select your Prometheus datasource and click "Import" to complete the procedure. Grafana finishes importing the dashboard and displays it on the screen with metrics coming from your Prometheus data source, as shown in the next screen capture:

At this point, you can use your monitoring infrastructure with Prometheus and Grafana to monitor the status and health of your Raspberry Pi data center.

Next Steps

Your basic IT infrastructure is ready to use. Using containers, you can run an entire data center's worth of applications to support your future projects. You can manage source code using Git with Gitea; you have a monitoring infrastructure ready in Prometheus, and a professional data visualization tool with Grafana. By using the node exporter and a pre-defined dashboard, you're already able to monitor your Raspberry Pi.

You'll use this same infrastructure in the upcoming home automation projects, using a similar approach you used to obtain monitoring data from your Pi. The basic workflow consists of exporting data from your applications, configuring Prometheus to scrape the metrics, adding a dashboard in Grafana to view the data, and potentially alerting on it.

Now that you have your personal data center running on your Raspberry Pi, you're ready to develop automation and monitoring programs, and then deploy them to this Infrastructure in a Box Pi hardware. In the next chapter, we'll do just that by building a temperature monitor to send alerts anytime it gets too hot (or too cold) in a room, a freezer, or even outdoors on a deck or porch. Let's go!

FOOTNOTES

[37]<https://docs.gitea.io/>

[38]<https://prometheus.io/>

[39]<https://prometheus.io/docs/prometheus/latest/configuration/configuration/>

[40]<https://grafana.com/oss/grafana/>

[41]<https://github.com/rfrail3/grafana-dashboards/blob/master/prometheus/node-exporter-full.json>

[42]<https://github.com/rfrail3/grafana-dashboards/blob/master/prometheus/node-exporter-full.json>

Part 2

Projects

Now that you have met all the project hardware and software requirements and have the infrastructure to keep your programs up and running, we're ready to begin building home automation solutions that use the Go language to make the magic happen.

In the four fun projects in this part, you'll explore important concepts that you can use to design and develop solutions for your own requirements: Tiny Go, GPIO, APIs, and using Pi Camera.

Chapter 4

Networking a Temperature Monitor

Now that you have your IT Infrastructure in a Box configured and ready to accept inbound data, you can begin building your first home automation project: a networked temperature monitor. This project will use the Raspberry Pi Pico W's onboard temperature sensor to report the current ambient temperature around the sensor. The Pico W will then communicate with the Raspberry Pi that is running the Prometheus server that you set up previously to poll a web server running on the Pico W. This web server will provide these temperature values in both Celsius and Fahrenheit measurement metrics.

Project's Hardware Requirements

This project requires these components:

- *Raspberry Pi Pico W*: The microcontroller with onboard Wi-Fi and a temperature sensor to report ambient temperature.
- *Raspberry Pi server*: A Raspberry Pi 3, 4, or Zero 2 to run the Prometheus exporter to scrape data from the Pico W.

For more details consult [Selecting a Raspberry Pi](#).

Once the sensor is calibrated and reporting its results via a REST-accessible JSON payload, you can deploy the Pico W in a number of environments such as a basement, a freezer, or even outdoors in order to obtain a reoccurring series of temperature updates. The Raspberry Pi you set up with Prometheus and Grafana in the previous chapter will reach out and poll the Pico W at set intervals. The formatted JSON data obtained from the Pico W will then be consumed by your Prometheus instance and visualized by your Grafana instance on the Raspberry Pi server. By visualizing the data, it'll be easy to spot trends and changes in temperature, as well as assign alerts when defined thresholds are exceeded. For example, you can configure Grafana to email you when your freezer temperature goes higher than 0 degrees Celsius (32 degrees Fahrenheit). That alert could save you from spoiling a lot of frozen food!

Remarkably, there's also work currently underway to make a minimized version of the Go language, called TinyGo^[43] that's capable of running on the Pi Pico W as well. At the time of writing this book, the Wi-Fi driver for the Pico W is not officially available with TinyGo, but the development version cyw43439^[44] is available. We'll use this driver for the book's project. When the driver is officially integrated within TinyGo,

the code will likely not change dramatically, but the driver's import path might change.

By the end of this chapter, you'll understand how to work with the Pico W device, and how to use TinyGo to develop Go programs that fit many microcontrollers. These important skills will allow you to develop other projects that use microcontrollers to handle hardware, connectivity, and logic in places or circumstances where it's harder to use larger, more power hungry, devices.

Now it's time to roll up your sleeves, gather up the necessary hardware, and start building the solution!

Understanding the Pico W Device

The Raspberry Pi Pico W is a Wi-Fi-enabled version of their popular Pico microcontroller design. Unlike the Raspberry Pi which runs a full Linux operating system and can be reached via a variety of network protocol connections, such as SSH, the Pico W is a microcontroller, and thus does not have the storage, memory, or operating system capacity of a Linux distribution. Instead, it's designed to immediately start a runtime at power on, and execute whatever program and script it's instructed to do. The Pico W currently supports its own C++ libraries and a minimal variant of Go called TinyGo. ^[45]

We'll take advantage of this wireless network-enabled Pico W version so that it can be placed anywhere there's a Wi-Fi signal it can associate with, and a power supply it can connect to. However, because the Pico W is a microcontroller and not a full-blown Linux-based server, we have to do some additional work to connect to and program for it.

The ideal development environment for a Pico W board is a desktop PC or laptop running Linux, macOS, or Windows, and a USB to USB micro cable to connect the Pico W to the computer. You can develop TinyGo programs using your preferred IDE or text editor. When you're done, you'll use the `tinygo` command-

line application to compile your program into a [UF2 firmware image](#) compatible with the Pico W. Install TinyGo on your development machine by following the instructions on the official Quick Install Guide. [\[46\]](#)

To transfer this image to the Pico W device, you need to start the Pico W in file transfer mode by holding down the white [BOOTSEL](#) button on the Pi Pico W while plugging in a USB cable between the Pico W and your PC, as shown in the next photo.

If successful, the PC should see the Pico W as a mountable USB drive, as indicated in the following screen capture.

Drag the freshly compiled UF2 file into the mounted Pico W's USB drive. The Pico W will automatically recognize this as a special file type by installing the file and rebooting the Pico W (and thereby ejecting its mounted USB drive in the process).

If everything goes well, your program will execute on the Pico W automatically. You can use TinyGo's `monitor` subcommand to monitor the Pico W serial interface for logs to ensure the program is running.

Now that you understand how to develop and transfer Go programs to the Pi Pico W, you can begin writing a REST server that will poll the Pico W's onboard temperature and report that

value in both Celsius and Fahrenheit, formatted in a JSON payload that can be consumed for further analysis. The values in this JSON will be converted into Prometheus-friendly formatting using a Go program that we'll write to perform the polling and conversion. But first, we need to get the temperature value off the Pico W's onboard temperature sensor, format it into JSON-friendly format, and have a simple HTTP server ready to accept new connections and deliver the JSON payload.

Polling the Temperature

In addition to having an onboard LED that's useful for visually indicating events and key activities, the Pico W also has an onboard temperature sensor. This sensor can be polled on a scheduled basis to determine the ambient temperature in the environment where the Pico W is operating.

Because the temperature sensor is onboard, it's close to the Pico W's CPU and other components, and the temperature sensor values may not be entirely accurate since it'll also sense any additional heat coming from the Pico W's electronics. This heat fluctuation can have a minor impact on measuring the actual room temperature. However, even if the actual temperature is slightly lower than what the onboard temperature sensor is recording, the measurement should be good enough for our monitoring purposes. That said, this should offset this increased value by estimating what to subtract from the captured onboard temperature reading.

TinyGo provides an abstracted library named `machine` to interact with components of different microcontroller devices. This abstraction makes it easier to reuse code across different devices sharing common functionality. This library also implements functions for features specific to some devices. For

the Pico W, `machine` implements the function `ReadTemperature` that performs a one-time read of the internal temperature sensor and outputs a value in milli-celsius. The following code snippet calculates both Celsius and Fahrenheit values depending on your reporting preference. Later, you'll incorporate this snippet into the larger Go web server application that will be created and deployed onto the Pico W.

```
func getTemperature() *temp {
    curTemp := machine.ReadTemperature()

    return &temp{
        TempC: float64(curTemp) / 1000,
        TempF: ((float64(curTemp) / 1000) * 9 / 5)
+ 32,
    }
}
```

Internally, the Raspberry Pi Pico W calculates the temperature by measuring the voltage on the fifth channel of the ADC (Analog-Digital Converter) controller. The function performs a conversion to output the value in milli-celsius.

```
func ReadTemperature() (millicelsius int32) {
    if rp.ADC.CS.Get() & rp.ADC_CS_EN == 0 {
```



```

        InitADC()
    }

    // Enable temperature sensor bias source
    rp.ADC.CS.SetBits(rp.ADC_CS_TS_EN)

    //  $T = 27 - (ADC\_voltage - 0.706) / 0.001721$ 
    return (27000<<16 - (int32(
        adcTempSensor.getVoltage()) - 706<<16) * 581)
    >> 16
}

```

You can find more details on how the temperature sensor works on the Raspberry Pi Pico RP2040 datasheet.^[47]

Now that you understand how to read the temperature, let's connect the Pico W to the Wi-Fi network.

Connecting the Pico W to Wi-Fi

To connect the Pico W to the Wi-Fi network using TinyGo, we'll use the open source driver developed by Patricio Whittingslow and Scott Feldman, and available at the [cyw43439 repository](#).^[48] In addition to the Wi-Fi driver, this repository also provides example boilerplate code to set up the connection and obtain an IP address automatically from the DHCP server. To use this boilerplate code, we need to download this repository to input the Wi-Fi credentials to connect to the network.

Let's start by creating a directory for this project:

```
$ mkdir -p tempmonitor/picoserver
$ cd tempmonitor
```

Then clone the [cyw43439](#) repository in this directory using [git](#):

```
$ git clone https://github.com/soypat/cyw43439
```

Next, switch to the cloned repository's [example/common](#) directory and copy the template credentials file into [secrets.go](#):

```
$ cd cyw43439/examples/common
$ cp secrets.go.template secrets.go
```

Then, edit the `secrets.go` file and specify your Wi-Fi connection details:

```
// Copy this file to secrets.go and make local  
changes to set Wifi credentials  
package common  
  
const (  
    // Set your Wifi SSID and passphrase here  
    ssid = "youWifiSSID"  
    pass = "yourPassword"  
)
```

Save and close this file. To ensure your program uses this version of the package, with your Wi-Fi credentials, let's ensure the Go module is correctly set up. Switch into the directory where the `picoserver` will reside:

```
$ cd ../../../../picoserver
```

Initialize a new Go module for your program:

```
$ go mod init picoserver  
go: creating new go.mod: module picoserver
```

This command creates a new `go.mod` file but it has no dependencies yet. Let's add the Pico W Wi-Fi driver as a dependency:

```
$ go mod edit \
    --require github.com/soypat/cyw43439@v0.0.0-
20240321235513-d28d7f302509
```

Finally, update the dependency to use the local repository which contains your Wi-Fi credentials:

```
$ go mod edit --
replace=github.com/soypat/cyw43439=./cyw43439
```

This is the content of the `go.mod` file:

```
module picoserver
```

```
go 1.22.1
```

```
require github.com/soypat/cyw43439 v0.0.0-
20240321235513-d28d7f302509
```

```
replace github.com/soypat/cyw43439 => ../cyw43439
```

Now that you have the correct dependency and the Wi-Fi credentials set up, let's start coding the temperature server.

Creating the Pico W REST Server

With the program's core components completed, there needs to be a way to have the Prometheus server remotely poll the Pico W's recorded temperature measurements so these metrics can be visualized on the Grafana instance that was set up previously. Doing so requires setting up a network connection using the onboard Wi-Fi adapter, associating it with your preferred Wi-Fi network's SSID and related password, and finally creating a simple HTTP server that will deliver the rounded `celsius` and `fahrenheit` values as a JSON payload.

Start your program by adding the package definition and the `import` statement to `main.go`:

[tempmonitor/picoserver/main.go](#)

```
package main

import (
    "bufio"
    "encoding/json"
    "io"
    "machine"
    "net/netip"
    "time"
```

```
"log/slog"  
  
"github.com/soypat/cyw43439"  
"github.com/soypat/cyw43439/examples/common"  
  
"github.com/soypat/seqs/httpx"  
"github.com/soypat/seqs/stacks"  
)
```

For this program, you're importing several external libraries including `machine` which is TinyGo's abstraction for the Pico W device, `log/slog` which is the new standard library for structured logs, `cyw43439` which contains the Pico W Wi-Fi driver, and `seqs/httpx` and `seqs/stacks` that provide a TCP stack and HTTP handling for the Pico W.

Next, define some constants that will be used to initialize the TCP stack:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```
const (  
    connTimeout = 3 * time.Second  
    maxconns    = 3  
    tcpbufsize  = 2030
```

```
hostname    = "picotemp"  
listenPort  = 80  
)
```

Then, define a new **struct** type that you'll use to convert the temperature obtained from the temperature sensor into JSON:

[tempmonitor/picoserver/main.go](#)

```
type temp struct {  
    TempC float64 `json:"tempC"`  
    TempF float64 `json:"tempF"`  
}
```

Next, define a package scoped variable of type pointer to **slog.Logger** that will serve as the default structured logger for the entire application. While using packages scoped variables is not encouraged, it's useful for a small device like the Pico W where there could be memory limitations if using many loggers. In addition, the new structured log type is a good choice here because it minimizes memory allocations for logging:

[tempmonitor/picoserver/main.go](#)

```
var logger *slog.Logger
```


Use Go's `init` function to initialize the default logger, redirecting the logging output to the Pico W serial interface by using the `machine.Serial` type as the first parameter. This will allow you to monitor applications logs later using the `tinygo monitor` command:

[tempmonitor/picoserver/main.go](#)

```
func init() {
    logger = slog.New(
        slog.NewTextHandler(machine.Serial,
            &slog.HandlerOptions{
                Level: slog.LevelInfo,
            })
    )
}
```

Next, create a function to change the Pico W LED state. You'll use this function to enable the Pico W LED to show that the device is active and listening for connections:

[tempmonitor/picoserver/main.go](#)

```
func changeLEDState(dev *cyw43439.Device, state
bool) {
    if err := dev.GPIOSet(0, state); err != nil {
        logger.Error("failed to change LED state:"
,
            slog.String("err", err.Error()))
    }
}
```

```
}  
  
}
```

Then use the [cyw43439](#) Wi-Fi driver and the example code available in the [examples/common](#) directory in the [cyw43439](#) repository^[49] to define a function that performs the Pico W Wi-Fi setup:

[tempmonitor/picoserver/main.go](#)

```
func setupDevice() (*stacks.PortStack,  
*cyw43439.Device) {  
    _, stack, dev, err :=  
common.SetupWithDHCP(common.SetupConfig{  
    Hostname: hostname,  
    Logger:   logger,  
    TCPPorts: 1,  
}))  
  
if err != nil {  
    panic("setup DHCP:" + err.Error())  
}  
  
    // Turn LED on  
    changeLEDState(dev, true)
```

```
    return stack, dev
}
```

Next, use the [seqs/stack](#) package to define a function that listens to the incoming TCP connection on port 80, as previously defined in the [constant](#) section:

[tempmonitor/picoserver/main.go](#)

```
func newListener(stack *stacks.PortStack)
*stacks.TCPListener {
    // Start TCP server.
    listenAddr := netip.AddrPortFrom(stack.Addr(),
listenPort)
    listener, err := stacks.NewTCPListener(
        stack, stacks.TCPListenerConfig{
            MaxConnections: maxconns,
            ConnTxBufSize:  tcpbufsize,
            ConnRxBufSize:  tcpbufsize,
        })
    if err != nil {
        panic("listener create:" + err.Error())
    }
    err = listener.StartListening(listenPort)
    if err != nil {
```

```

        panic("listener start:" + err.Error())
    }

    logger.Info("listening",
        slog.String("addr", "http://"+listenAddr.String()),
    )

    return listener
}

```

Add a function to blink the LED. You'll use this function to provide visual feedback when the Pico W server receives a connection request. This function uses a Go **channel**, as it'll run concurrently with other functions to avoid blocking the program:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```

func blinkLED(dev *cyw43439.Device, blink chan
uint) {
    for {
        select {
        case n := <-blink:
            lastLedState := true

```

```

        if n == 0 {
            n = 5
        }

        for i := uint(0); i < n; i++ {
            lastLedState = !lastLedState
            changeLEDState(dev, lastLedState)
            time.Sleep(500 * time.Millisecond)
        }

        // Ensure LED is on at the end
        changeLEDState(dev, true)
    }
}

```

Then, add the function to obtain the temperature from the temperature sensor and output the values in both Celsius and Fahrenheit using the `temp` custom type you defined previously:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```

func getTemperature() *temp {
    curTemp := machine.ReadTemperature()

    return &temp{
        TempC: float64(curTemp) / 1000,
    }
}

```

```
        TempF: ((float64(curTemp) / 1000) * 9 / 5)
+ 32,
    }
}
```

Next, define the function `HTTPHandler` to handle the HTTP request for temperature. This function uses the `segs/httpx` library to define HTTP headers for the response. In the function body, obtain the temperature from the sensor using `getTemperature` and convert it to JSON using the standard library `encoding/json` package. In case of errors, return a 500 (Internal Server Error) response. In case of success, return the JSON containing the temperature:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```
func HTTPHandler(respWriter io.Writer, resp
*httpx.ResponseHeader) {
    resp.SetConnectionClose()
    logger.Info("Got temperature request...")
    t := getTemperature()

    body, err := json.Marshal(t)
    if err != nil {
        logger.Error(
```

```

        "temperature json:",
        slog.String("err", err.Error()),
    )
    resp.SetStatusCode(500)
} else {
    resp.SetContentType("application/json")
    resp.SetContentLength(len(body))
}
respWriter.Write(resp.Header())
respWriter.Write(body)
}

```

Write a function to handle HTTP connections and respond with the temperature JSON. Again, because this is a small device, define some buffer that can be reused for all connections to avoid memory allocations. This function also takes a channel as input. This channel notifies the `blinkLED` goroutine to blink the Pico W LED, showing that it's processing a connection:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```

func handleConnection(listener
*stacks.TCPLListener, blink chan uint) {
    // Reuse the same buffers for each
    // connection to avoid heap allocations.

```

```
var resp httpx.ResponseHeader
buf := bufio.NewReaderSize(nil, 1024)

for {
    conn, err := listener.Accept()
    if err != nil {
        logger.Error(
            "listener accept:",
            slog.String("err", err.Error()),
        )
        time.Sleep(time.Second)
        continue
    }
}
```

```
logger.Info(
    "new connection",
    slog.String("remote",
        conn.RemoteAddr().String()),
)
err =
conn.SetDeadline(time.Now().Add(connTimeout))
if err != nil {
    conn.Close()
    logger.Error(
```



```

        "conn set deadline:",
        slog.String("err", err.Error()),
    )
    continue
}
buf.Reset(conn)
resp.Reset()
HTTPHandler(conn, &resp)
conn.Close()

blink <- 5
}
}

```

Finally, put everything together in the `main` function that serves as the program's entry point:

[temppmonitor/picoserver/main.go](https://github.com/temppmonitor/picoserver/main.go)

```

func main() {
    stack, dev := setupDevice()
    listener := newListener(stack)

    blink := make(chan uint, 3)
    go blinkLED(dev, blink)
}

```

```
go handleConnection(listener, blink)

for {
    select {
        case <-time.After(1 * time.Minute):
            logger.Info("Waiting for
connections...")
    }
}
}
```

The `main` function sets the Wi-Fi connection up, creates the TCP listener, defines the `blink` channel to connect the `blinkLED` and `handleConnection` goroutines, and starts both goroutines to process requests. In the end, it uses an infinite loop with a blocking `select` statement to prevent the program from terminating while the goroutines run in the background. It also sends a message every minute to the log to show the program is still running.

While the Pico W does have a fast enough processor to keep up with simple web server connection requests, it can become quickly overwhelmed with many simultaneous requests, or by timing out while attempting to process a very large JSON payload. However, because the two values being returned are

fairly small and lightweight, you shouldn't encounter these limitations as long as you don't have more than one client calling upon this server once every several seconds.

Save the `main.go` file and run `go mod tidy` to download the required dependencies:

```
$ go mod tidy
```

Now, use the `tinygo` command line to build your program:

```
$ tinygo build -target=pico -opt=1 -stack-size=8kb  
-size=short -o main.uf2 .  
  
   code      data      bss |   flash      ram  
594516    16116     3440 | 610632    19556
```

Then copy the resulting `uf2` file to your Pico W by dragging and dropping it to the mounted device, or using the command line:

```
$ cp main.uf2 /mnt/RPI-RP2
```

The Pico W will restart automatically and start the program. Use `tinygo monitor` to see the logs on the Pico W serial interface:

```
$ tinygo monitor
```

```
Connected to /dev/ttyACM0. Press Ctrl-C to exit.
```

time=1970-01-01T00:00:05.003Z level=INFO msg="wifi
join success!"

mac=28:cd:c1:d1:c0:52

time=1970-01-01T00:00:05.006Z level=INFO msg="DHCP
ongoing..."

time=1970-01-01T00:00:05.008Z level=INFO
msg=DHCP:tx msg=Discover

time=1970-01-01T00:00:05.058Z level=INFO
msg=DHCP:rx msg=Offer

time=1970-01-01T00:00:05.059Z level=INFO
msg=DHCP:tx msg=Request

time=1970-01-01T00:00:05.061Z level=INFO
msg=DHCP:rx msg=Offer

time=1970-01-01T00:00:05.109Z level=INFO
msg=DHCP:rx msg=Offer

time=1970-01-01T00:00:05.160Z level=INFO
msg=DHCP:rx msg=Offer

time=1970-01-01T00:00:05.211Z level=INFO
msg=DHCP:rx msg=Ack

time=1970-01-01T00:00:05.262Z level=INFO
msg=DHCP:rx msg=Ack

time=1970-01-01T00:00:05.507Z level=INFO msg="DHCP
complete"

```
cidrbits=24 ourIP=192.168.30.180
dns=192.168.30.1
broadcast=192.168.30.255 gateway="invalid
IP"
router=192.168.30.1 dhcp=192.168.30.1
hostname=picotemp
lease=24h0m0s renewal=12h0m0s
rebinding=21h0m0s
time=1970-01-01T00:00:05.512Z level=INFO
msg=lst:freeConnForReuse lport=0 rport=0
time=1970-01-01T00:00:05.513Z level=INFO
msg=lst:freeConnForReuse lport=0 rport=0
time=1970-01-01T00:00:05.514Z level=INFO
msg=lst:freeConnForReuse lport=0 rport=0
time=1970-01-01T00:00:05.515Z level=INFO
msg=listening
addr=http://192.168.10.180:80
time=1970-01-01T00:00:16.403Z level=INFO
msg=TCP:rx-statechange port=80 old=Listen
new=SynRcvd rxflags=[SYN]
time=1970-01-01T00:00:16.713Z level=INFO
time=1970-01-01T00:00:16.714Z level=INFO
msg="new connection"
remote=192.168.30.122:59588
```

```
time=1970-01-01T00:00:16.715Z level=INFO msg="Got
temperature request..."
```

The program may take several seconds to initialize and associate a network connection with your wireless network. If successful, you should see a printout of the IP address your router assigned to the Pico W. Then poll that IP address using `curl`. For example, if your Pico W's assigned IP address is `192.168.30.180`, use `curl http://192.168.30.180` like this:

```
$ curl http://192.168.30.180
{"tempC": 25.6, "tempF": 78.08}%
```

If you prefer, you can also use a browser or REST-specific API utilities, like Postman^[50] on the Desktop and API Tester^[51] on mobile devices, to verify and troubleshoot expected return results. However, since this is a simple read-only GET request, using `curl` is more than adequate for our needs.

If your test is successful, you can take the next bold step of unplugging the USB cable from your laptop and connecting it to a spare USB power supply. Relocate the Pico W attached to this dedicated power source, and connect it to any power outlet within range of your assigned SSID broadcast. Within a minute or so, check to see if the IP is reachable via your web browser

test again. If the test fails, check the logs provided by running the `tinygo monitor` command to understand the failure cause and address it. Ensure your Pico W device is in range of your Wi-Fi router and the credentials are correct. If you cannot see any logs, transfer your program to the Pico W again to ensure it did not have any issues during the file transfer.

Now we're ready to move on to the next phase of the project, building the Prometheus exporter that will poll the Pico W's assigned IP address every ten seconds to capture the temperature data. Grafana will process the captured data to visualize the ambient temperature around the Pico W.

Creating the Prometheus Exporter

Now that a hardware endpoint exists to query at any time, use that always-on accessibility to your advantage by leveraging the power of the Prometheus server you installed in the last chapter. Prometheus uses its own particular naming conventions and formatting rules for indicating value names and related assignments. As such, you'll need to convert the JSON payload received by the Pico W server into an exported format that Prometheus can consume. Fortunately, this is a trivial process to create in Go because Prometheus itself is written in Go.

This implementation, while not sophisticated, applies several of Prometheus exporter development guidelines,^[52] ensuring it exports metrics according to Prometheus's conventions, and is safe for concurrent use.

Start by creating a directory for this project and switching to it:

```
$ mkdir -p gohome/picotempexport  
$ cd gohome/picotempexport
```

Then initialize a new Go module for the project:


```
$ go mod init gohome/picotempexport
```

Now, create an HTML file to hold the template for the root page, redirecting clients to the metrics page where they can obtain the metrics exported. Later, you'll use Go's `embed` package to embed this file into the final binary.

[tempmonitor/exporter/rootPage.html](#)

```
<html>
  <head>
    <title>Pi Pico W Temperature
Exporter</title>
  </head>
  <body>
    <h1>Pi Pico W Temperature Exporter</h1>
    <p><a href= '/metrics'>metrics</a></p>
  </body>
</html>
```

Next, open your favorite Go editor and create a new file, `picotempexport.go`. Add `package main`, followed by importing the native and third-party libraries needed. These include the Prometheus-specific libraries.

[tempmonitor/exporter/picotempexport.go](#)

```
package main
```

```
import (
    _ "embed"
    "encoding/json"
    "errors"
    "fmt"
    "log"
    "net/http"
    "os"
    "sync"
    "time"

    "github.com/prometheus/client_golang/prometheus"

    "github.com/prometheus/client_golang/prometheus/promhttp"

    "github.com/prometheus/client_golang/prometheus/promauto"
)
```

Now, use the `embed` module to embed the HTML content into the variable `rootPageHTML`. The `embed` module works as a compiler directive that instructs the compiler to create the variable defined in the next line, assigning its value to the content of the embedded file:

[tempmonitor/exporter/picotempexport.go](#)

```
//go:embed rootPage.html  
var rootPageHTML []byte
```

Then, create a variable encoding a custom error value `errInvalidResponse` that indicates the Pico W server returned invalid content:

[tempmonitor/exporter/picotempexport.go](#)

```
var errInvalidResponse = errors.New("unexpected  
response from the server")
```

Next, create a `struct` to hold the Celsius and Fahrenheit values parsed from the Pico W server's JSON payload.

[tempmonitor/exporter/picotempexport.go](#)

```
type tempValues struct {  
    TempC float64 `json:"tempC"`  
    TempF float64 `json:"tempF"`  
}
```

Now write the code for the `getTempValues()` method that will connect to the Pico W server, make a request, receive the data, and then parse the JSON payload into an instance of `tempValues` struct that you'll use later to return metrics to Prometheus:

[tempmonitor/exporter/picotempexport.go](#)

```
func (tv *tempValues) getTempValues(client
*http.Client, url string) error {
    response, err := client.Get(url)
    if err != nil {
        log.Println(err)
        return err
    }
    defer response.Body.Close()
```

```
    if response.StatusCode != http.StatusOK {
        err := fmt.Errorf(
            "%w: invalid status code: %s",
            errInvalidResponse,
            response.Status,
        )
        log.Println(err)
        return err
    }

    if err :=
json.NewDecoder(response.Body).Decode(tv); err !=
nil {
    log.Println(err)
    return err
}
```

```
}  
  
    return nil  
  
}
```

Next, define another type, `metrics`, to cache the metrics values before sending the data to Prometheus. This new type accomplishes two important goals:

1. Defines a concurrency-safe model to access data in case more than one instance of Prometheus requests them simultaneously.
2. Caches the values received from the Pico W server for two seconds, preventing overloading the Pico W in case the exporter receives many concurrent requests.

The `metrics` struct includes an instance of struct `tempValues` to store the metrics received from the Pico W server, and a field `up` that indicates when the Pico W is up and running. It also includes an `expire` field representing a time when the cache expires and it must obtain new values from the Pico W server. Finally, it embeds type `sync.RWMutex` from the standard library `sync`, which allows locking and unlocking the struct for safe concurrent access. Define the new struct like this:

[temponitor/exporter/picotempexport.go](https://github.com/temponitor/exporter/picotempexport.go)

```
type metrics struct {  
    results *tempValues  
    up      float64  
    expire  time.Time  
    sync.RWMutex  
}
```

Then, associate a method `getMetrics` to the `metrics` type. This method returns the existing values if the cache hasn't expired, or if the cache has expired, it uses the previously defined method `getTempValues` to obtain new temperature values from the Pico W server:

tempmonitor/exporter/picotempexport.go

```
func (m *metrics) getMetrics(client *http.Client,  
url string) *metrics {  
    m.Lock()  
    defer m.Unlock()  
  
    if time.Now().Before(m.expire) {  
        return m  
    }  
  
    m.up = 1
```

```

    if err := m.results.getTempValues(client,
url); err != nil {
        m.up = 0
        m.results.TempC = 0
        m.results.TempF = 0
    }

    m.expire = time.Now().Add(2 * time.Second)

    return m
}

```

Next, define a set of methods to access the metrics values in a concurrent safe way, by locking the struct instance for reading before returning their values, and unlocking it when it's done:

[temponitor/exporter/picotempexport.go](https://github.com/temponitor/exporter/picotempexport.go)

```

func (m *metrics) tempC() float64 {
    m.RLock()

    defer m.RUnlock()

    return m.results.TempC
}

```

```

func (m *metrics) tempF() float64 {

```

```
m.RLock()  
  
defer m.RUnlock()  
  
return m.results.TempF  
}
```

```
func (m *metrics) status() float64 {  
    m.RLock()  
    defer m.RUnlock()  
  
    return m.up  
}
```

The Prometheus exporter exports metrics over an HTTP interface. You will accomplish it by writing a small HTTP server using Go's standard `net/http` and Prometheus's `promhttp` packages to expose metrics. First, create a new HTTP requests multiplexer to handle incoming connections and dispatch them to the appropriate handler. This function uses Prometheus's package `promauto` to define the three target metrics that you're exposing in a Prometheus standard format:

[tempmonitor/exporter/picotempexport.go](https://github.com/tempmonitor/exporter/picotempexport.go)

```
func newMux(url string) http.Handler {  
    mux := http.NewServeMux()
```



```

client := &http.Client{
    Timeout: 10 * time.Second,
}

m := &metrics{
    results: &tempValues{},
}

promauto.NewGaugeFunc(
    prometheus.GaugeOpts{
        Name:        "pico_temperature",
        Help:        "Pico Sensor
Temperature.",
        ConstLabels: prometheus.Labels{"unit":
"celsius"},
    },
    func() float64 {
        return m.getMetrics(client,
url).tempC()
    },
)

promauto.NewGaugeFunc(

```

```
    prometheus.GaugeOpts{
        Name:      "pico_temperature",
        Help:      "Pico Sensor
Temperature.",
        ConstLabels: prometheus.Labels{"unit":
"fahrenheit"},
    },
    func() float64 {
        return m.getMetrics(client,
url).tempF()
    },
)

promauto.NewGaugeFunc(
    prometheus.GaugeOpts{
        Name: "pico_up",
        Help: "Pico Sensor Server Status.",
    },
    func() float64 {
        return m.getMetrics(client,
url).status()
    },
)
```

```
    mux.HandleFunc("/", func(w
http.ResponseWriter, _ *http.Request) {
        w.Write(rootPageHTML)
    })

    mux.Handle("/metrics", promhttp.Handler())

    return mux
}
```

Note that Prometheus defaults to the `/metrics` absolute path when polling a server if the endpoint isn't explicitly defined. While not essential, it's a good practice to include a brief description of the endpoint at the root level of the web server. This multiplexer encodes that practice by handling requests to the server root `/` with a snippet of HTML that redirects to `/metric`.

Finally, create a `main()` function that provides an entry point to the program and starts the HTTP server on port `:3030` using the multiplexer you previously defined. This function also captures the Pico W server URL using an environment variable enabling you to customize it when starting the program, and a custom HTTP server object that defines some sensible timeouts to prevent connections from hanging forever in case of issues.

While this is not required to develop a web server with Go, it's a recommended practice. Define the main function like this:

[tempmonitor/exporter/picotempexport.go](#)

```
func main() {
    picoURL := os.Getenv("PICO_SERVER_URL")

    s := &http.Server{
        Addr:           ":3030",
        Handler:        newMux(picoURL),
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
    }

    if err := s.ListenAndServe(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

With all the code in place, and the defined Pico W REST server up and running, compile and run this Go Prometheus exporter using the typical `go run` command. Set the environment variable `PICO_SERVER_URL` to your Pico W server URL to configure the server to obtain temperatures from your Pico W.

```
$ PICO_SERVER_URL=http://<PICO_IP_OR_HOSTNAME> go  
run picotempexport.go
```

Assuming no errors, you should be able to open a browser on your local machine and visit <http://localhost:3030> with your preferred web browser. If successful, you should see something similar to the response in the next screenshot.

Selecting the [/metrics](#) link on the page should show the Prometheus-formatted results, including the defined [pico_temperature](#) with both Celsius and Fahrenheit labels and associated values.

```
# HELP pico_temperature Pico Sensor Temperature.  
# TYPE pico_temperature gauge  
pico_temperature{unit="celsius"} 31  
pico_temperature{unit="fahrenheit"} 87.8  
# HELP pico_up Pico Sensor Server Status.  
# TYPE pico_up gauge  
pico_up 1
```

Congratulations! You just attained another milestone by converting the Pico W onboard temperature values into a format that can be consumed by the Prometheus server. But before you can configure the Prometheus server to perform a

scheduled polling of this exporter, you need to package it into a container, and then deploy and run this container in Docker so you know what URL to add to Prometheus's polling configuration.

Containing and Deploying the Exporter

Now that you have a working exporter for your Pico W temperature sensor, let's create an image to run the exporter as a container in your Raspberry Pi infrastructure, similar to what you did in [*Containing the Server*](#).

First, create a Dockerfile using the multi-stage approach to compile your Go Prometheus exporter in the first stage, and create an image for it in the second stage:

[tempmonitor/exporter/Dockerfile](#)

```
FROM docker.io/golang:1.22 AS builder
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build
  -ldflags="-s -w"

FROM docker.io/alpine:latest
RUN mkdir /app && adduser -h /app -D
  picotempexport
WORKDIR /app
COPY --chown=picotempexport --from=builder
  /app/picotempexport .
```

```
EXPOSE 3030
CMD ["/app/picotempexport"]
```

Then, build the image by using command `docker build`:

```
$ docker build -t picotempexport:v1 .
```

Verify the command worked and built the image successfully by listing local images on your Raspberry Pi:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
			SIZE
picotempexport	v1	d1fc052c1ee0	12 seconds ago
			13.8 MB

Test your image by running the export with `docker`. Set the environment variable `PICO_SERVER_URL` to your Pico W server IP or hostname, and expose port `3030` for external connection. Use `--restart=always` to ensure the container starts automatically on system startup. Attach the container to the Prometheus server network `prometheus_prom_net` so Prometheus can scrape this container for metrics:

```
$ docker run -d \
    --name picotempexport-v1 \
```



```
-p 3030:3030 \  
--env  
PICO_SERVER_URL=http://<PICO_IP_OR_HOSTNAME> \  
--restart=always \  
--net=prometheus_prom_net \  
picotempexport:v1
```

Now, verify the container is running, using `docker ps -l` to list the last container created:

```
$ docker ps -l  
CONTAINER ID    IMAGE                                     COMMAND  
CREATED  
STATUS          PORTS  
NAMES  
e30d7e6e64d6    picotempexport:v1  
/app/picotempexpo... About a minute ago  
Up About a minute ago    0.0.0.0:3030->3030/tcp  
picotempexport-v1
```

You can also test your exporter by using `curl` to query metrics on the exposed port `3030`:

```
$ curl -s http://localhost:3030/metrics | grep  
pico
```

```
# HELP pico_temperature Pico Sensor Temperature.
# TYPE pico_temperature gauge
pico_temperature{unit="celsius"} 59
pico_temperature{unit="fahrenheit"} 138.2
# HELP pico_up Pico Sensor Server Status.
# TYPE pico_up gauge
pico_up 1
```

Your Prometheus exporter is ready and exporting temperatures collected from the Pico W. In the next step, you'll configure Prometheus to scrape these metrics.

Configuring Prometheus to Query the Exporter

Thanks to the service discovery job you implemented with your Prometheus configuration, you can use the same approach you used in [*Monitoring and Alerting with Prometheus*](#), to scrape metrics from the Node exporter, to configure your custom exporter. Since the custom exporter container is running on the same Raspberry Pi where Prometheus service runs and you attached it to the same network `prometheus_prom_net`, you can use the container name `picotempexport-v1` 3030 to obtain the metrics:

[`tempmonitor/prometheus/sd_picotempexporter.yml`](#)

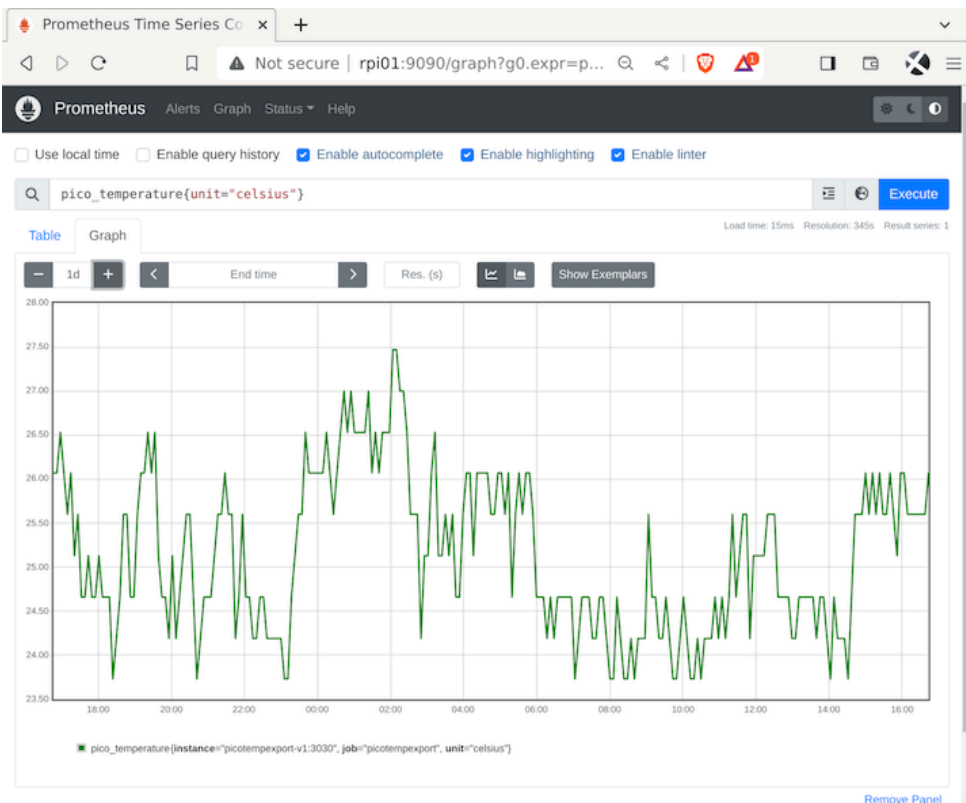
```
- labels:  
  job: picotempexport  
targets:  
  - 'picotempexport-v1:3030'
```

Then, use the command `docker cp` to copy the service discovery file to the Prometheus container:

```
$ docker cp sd_picotempexporter.yml prometheus-  
prometheus-1:/prometheus
```

Wait a few seconds and refresh the Prometheus Targets page to see the new target automatically included, as demonstrated in the next screenshot:

Finally, navigate to the `Graph` tab and select `pico_temperature` metric to view current values in Celsius and Fahrenheit. You can also add a label `{unit="celsius"}` to display metrics in one of the units only. Select the `Graph` sub tab to view the temperature variation over time, as shown in the [screenshot](#).



You have successfully configured your Prometheus environment to collect metrics from your Pico W. In the next step, you'll use Grafana to create a nice temperature-over-time display.

Creating the Grafana Dashboard

Assuming you already have a running Grafana instance from [*Visualizing Data with Grafana*](#), you already have Grafana set up to pull data from the Prometheus server that you just configured. This makes it easy to create a new Grafana dashboard, add a panel to that dashboard, and set the panel's metric to `pico_temperature` which was defined earlier.

To create a new Grafana dashboard, select the Dashboards icon from Grafana's icon toolbar running down the left upper corner of your Grafana instance. Then select the `New Dashboard` from the `New` drop-down menu, as shown in the next screen capture.

Selecting this option will generate a new, blank Grafana dashboard, and prompt you to add a new visualization, as shown in the next screenshot.

Select the `Add visualization` option. This will generate a new panel within the dashboard, allowing you to identify the metric to display in the panel, along with the type of chart to visualize the data. Make sure that your Prometheus server is selected as the data source, and then in the first Query location (labeled `A`), enter `pico_temperature` as the metric to visualize. Decide which unit

(Celsius or Fahrenheit) to display in the time series chart and set it accordingly. For example, if you want to see the `pico_temperature` graph results in Fahrenheit, select `unit` in the `Labels` section, and set it equal to `fahrenheit`. This setup should look similar to the next screen image.

On the right side of this screen, you can choose a number of options, such as the type of chart (keep the default `Time series` for now), plot color, and panel label, to name a few. For example, if the Pico W was monitoring the temperature of a basement, you could label the panel `Basement Temperature`. Select the `Save` button in the upper right corner of the page to save your changes. Once saved, you should see the temperature data that the Prometheus server has collected so far, visualized in a time series chart.

Although you have essentially completed the objective, it's time to leverage the power of Grafana's alerting rules to generate an alert whenever a condition is met. In this case, configure Grafana to send an email any time the temperature being monitored drops below 66 degrees. Grafana supports a number of different messaging services, ranging from Discord, Slack, and Microsoft Teams integration, to more traditional SMS and email transmissions. Create your messaging option of choice by selecting the `Alerting` bell-shaped icon from Grafana's icon

toolbar. Then select **+ New Contact Point** and select email from the list. Note that you may need to configure and test your email server to work with Grafana before you can select it. Since email servers can be configured to integrate with Grafana in a variety of ways, visit [Configure Grafana](#)^[53] on Grafana's website for more details.

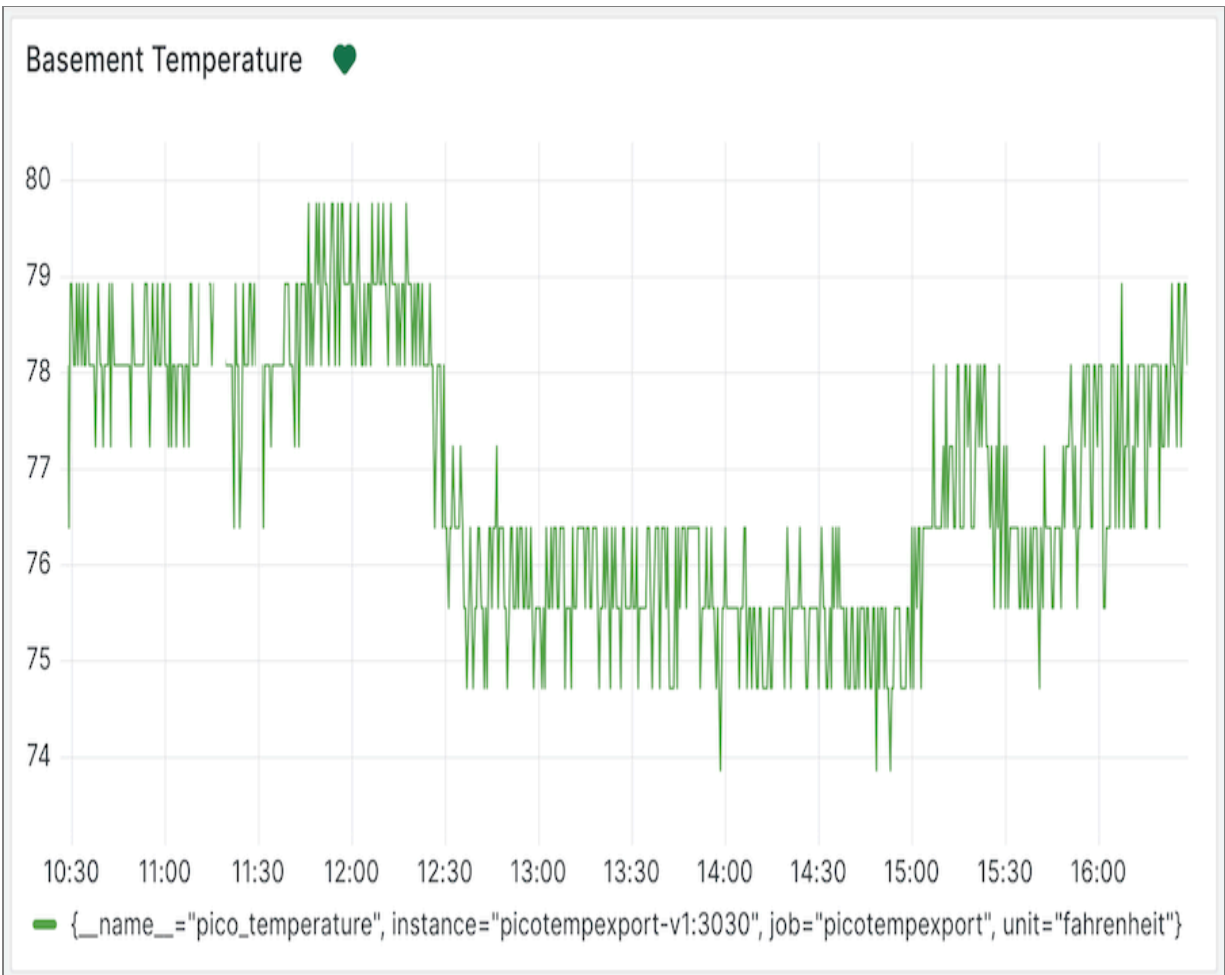
Once the default messaging service has been configured and tested, you can use it as the **Contact point** in this temperature alert rule. Edit the **Basement Temperature** panel by hovering the mouse cursor over the **Basement Temperature** label (or whatever other label you might have called your panel), and select the **Edit** option from the drop-down menu. Then select the **Alert** tab from the configuration page and click on **Create alert rule from this panel**. Since you want Grafana to send an email alert any time the temperature drops below, say, 66 degrees Fahrenheit, scroll down to the expression **C** area and set the threshold expression with the condition **INPUT B IS BELOW 66**. Grafana automatically created an expression **B** which represents the last value of **A** which in turn represents the metric we're measuring. These settings are shown in the [screenshot](#).

The screenshot displays the Grafana alert configuration interface. It features two expression configuration panels, B and C, stacked vertically. Panel B is titled 'Reduce' and includes a 'Function' dropdown set to 'Last', an 'Input' dropdown set to 'A', and a 'Mode' dropdown set to 'Strict'. Below these settings is a blue link that says 'Make this the alert condition'. Panel C is titled 'Threshold' and includes an 'Input' dropdown set to 'B', a comparison operator dropdown set to 'IS BELOW', and a value input field set to '66'. Below these settings is a green box with a checkmark and the text 'Alert condition'. At the bottom of the interface are three buttons: '+ Add query', '+ Add expression', and a blue 'Preview' button.

Scroll down to the [Alert Evaluation behavior](#) section and select a folder and evaluation group for your alert. If this is your first Grafana alert, type the names on the boxes and press [Enter](#) to create a new folder and group. Leave the remaining settings in this section as default. Doing so instructs Grafana to test the condition every minute for a duration of five minutes. This way, if the temperature oscillates between say, 66 and 67 degrees every other minute, your email inbox doesn't get overwhelmed with Grafana alerts. The result is shown in the next capture.

You can optionally add details and labels to your alert. When done, select the [Save and exit](#) button in the upper right corner of this alert configuration page, and after a few minutes, you

should see a green heart next to your panel label. This means that everything is healthy since the alerting rule has not been triggered because the threshold conditions have not been met. An example of this is shown in the [screen capture](#).



When the alert condition is triggered, you should receive an email alert from Grafana indicating the values that triggered the alert. When you visit the dashboard, you'll also see that Grafana has changed the heart from green and healthy to a

broken red heart. It's this type of nice visual flair that has endeared Grafana to IT pros and DevOps engineers alike.

The next photo shows the Pico W monitoring temperature next to a much more expensive, proprietary home automation temperature monitor. Even adding an inexpensive mini display to the Pico W would still keep it an order of magnitude cheaper than the commercial monitoring counterpart.

If you want to monitor more than one temperature scenario, simply add more Pico Ws to your network running the REST service. Then add this new Pico W's IP address to the exporter with the variables you want assigned to it and map these to the Grafana temperature panel you created. Then you can have an army of Pico Ws monitoring a variety of temperature collection tracking scenarios.

Next Steps

Congratulations! You built a sophisticated REST-based service that is being monitored using the same tools that large enterprises employ in their IT organizations. This robust configuration should maintain operational uptime as long as power continues to flow through your Pi hardware.

Now that you can monitor temperature anywhere within range of your wireless network, you can deploy the Pico W into a variety of scenarios. Want to monitor the temperature of your refrigerator or freezer? Place the Pico W into it and let it run. If outdoor temperature is more desired, house the Pico W in a weatherproof case and shield it from sunlight while sampling the degrees. Note that for those cases, you may want to power your Pico W using batteries as suggested in the official Pico W guide.^[54]

In our next project, you'll build upon the infrastructure and application principles created for the temperature monitor, and use a motion detection sensor to identify whenever something changes in the monitored environment. See you in the next chapter!

[43]<https://tinygo.org/>

[44]<https://github.com/soypat/cyw43439>

[45]<https://tinygo.org/>

[46]<https://tinygo.org/getting-started/install/>

[47]<https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>

[48]<https://github.com/soypat/cyw43439>

[49]<https://github.com/soypat/cyw43439/examples/common>

[50]<https://www.postman.com/>

[51]<https://apitester.org/>

[52]https://www.prometheus.io/docs/instrumenting/writing_exporters/

[53]<https://grafana.com/docs/grafana/latest/setup-grafana/configure-grafana/>

[54]<https://projects.raspberrypi.org/en/projects/introduction-to-the-pico/12>

Chapter 5

Checking the (Garage) Door

In Chapter 4, [*Networking a Temperature Monitor*](#), you used the Raspberry Pi Pico W's embedded temperature sensor to monitor room temperature. In this chapter, you'll combine the versatility of Raspberry Pi Zero 2W with a magnetic contact switch to determine if a door—in this case, a garage door—is open or closed. For more details about the differences between these two Raspberry Pi devices, consult [*Selecting a Raspberry Pi*](#).

You will also develop the software to report the garage door status, using an API endpoint similar to the one you developed in Chapter 2, [*Building a REST API Server*](#), and to send a Discord notification if you left the garage door open at night.

Project's Hardware Requirements

This project requires the following components:

- *Raspberry Pi Zero 2W*: A small and versatile device with GPIO pins to connect the magnetic switch, and run the required software.

- *Magnetic contact switch:* A small magnetic sensor that indicates if a door is open or closed.
- *2 jump wires:* to connect the sensor to the GPIO pins on the Pi Zero W.
- *Solderless Pin Headers:* to provide an easy alternative to enable the Pi Zero 2W GPIO pins without requiring solder, in case your Pi Zero does not come with GPIO pins.

For more details, consult [*Adding Other Hardware Components*](#).

The Raspberry Pi Zero is a remarkable device. Not only is it small and versatile, but it's also equipped with a General Purpose Input/Output (GPIO) interface. The GPIO allows you to expand the Raspberry Pi and connect it to other devices and sensors, providing the foundation to develop many other home automation projects. Some examples of how you can use the GPIO in home automation include:

- Using distance sensors to notify if someone or something approaches
- Using motion sensors to detect movement in an area, like the porch
- Controlling lights and power outlets
- Connecting with temperature or humidity sensors to control the environment
- Verifying if doors or windows are closed

- Controlling motors or fans (with the help of other devices)

Upon completing this project, you'll have both the hardware and software foundation to develop other projects that use the Raspberry Pi GPIO. Let's get started.

Understanding the GPIO

The GPIO is a generic term for input/output interfaces; their implementation varies across different vendors and devices. Usually vendors combine the GPIO circuitry with a row of pins which allow external access to the device. These pins are usually known as GPIO pinout or GPIO pins.

One of the major benefits of the Raspberry Pi GPIO implementation is the standardization of the GPIO pinout across different Raspberry Pi models. All modern Raspberry Pi devices—from the Raspberry Pi 3 to Raspberry Pi 5, and Raspberry Pi Zero—implement a 40 pin GPIO interface, arranged in two parallel rows of 20 pins. The pins in all these devices also share the same function; therefore, learning how to use the GPIO in one device allows you to use it on all these devices. Please note that the Raspberry Pi Pico and Pico W have a different pinout layout than other Raspberry Pi devices due to their smaller size and other constraints.

You can learn more about the Raspberry Pi's GPIO layout and pin functions on the official Raspberry Pi Physical Computing Guide. [\[55\]](#)

The Raspberry Pi GPIO has an embedded protection which makes connecting low-power devices such as LEDs or small sensors relatively safe. Keep in mind that it's still an electrical device, and connecting devices that use a lot of power may damage your Raspberry Pi. Do not connect motors, fans, or incandescent lamps to your Pi, for example.

You can configure the GPIO pins as input or output depending on your project's requirements. The Raspberry Pi implementation provides a logic circuit that allows you to configure the pins via software for increased convenience. This allows you to quickly convert a Pi from one project to another without tampering with circuits or jumpers, and it also allows you to use a single Pi for multiple projects simultaneously.

A final benefit of the Raspberry Pi GPIO implementation is an embedded Pull Up/Pull Down resistor you can activate via software when needed. A Pull Up/Pull Down resistor ensures a known state for a signal which allows you to quickly detect when the signal has changed and act on it. Without an embedded resistor like this, many projects would require an external circuit that provides the same capabilities. This convenience makes it easier to develop home automation projects using the Pi's GPIO. You'll use the embedded Pull Up resistor in this chapter's project.

Many Raspberry Pi models provide the GPIO pin pre-soldered onto the board and ready to use. Some smaller modes, such as the Raspberry Pi Zero 2W used in this chapter, may or may not come with pre-soldered pins. In case your Pi model does not have pins, we recommend using solderless pin headers that you can easily snap onto your Pi, unless you're comfortable soldering pins to your board.

Now that you have a basic understanding of the GPIO functionality, let's wire the magnetic sensor to your Raspberry Pi GPIO.

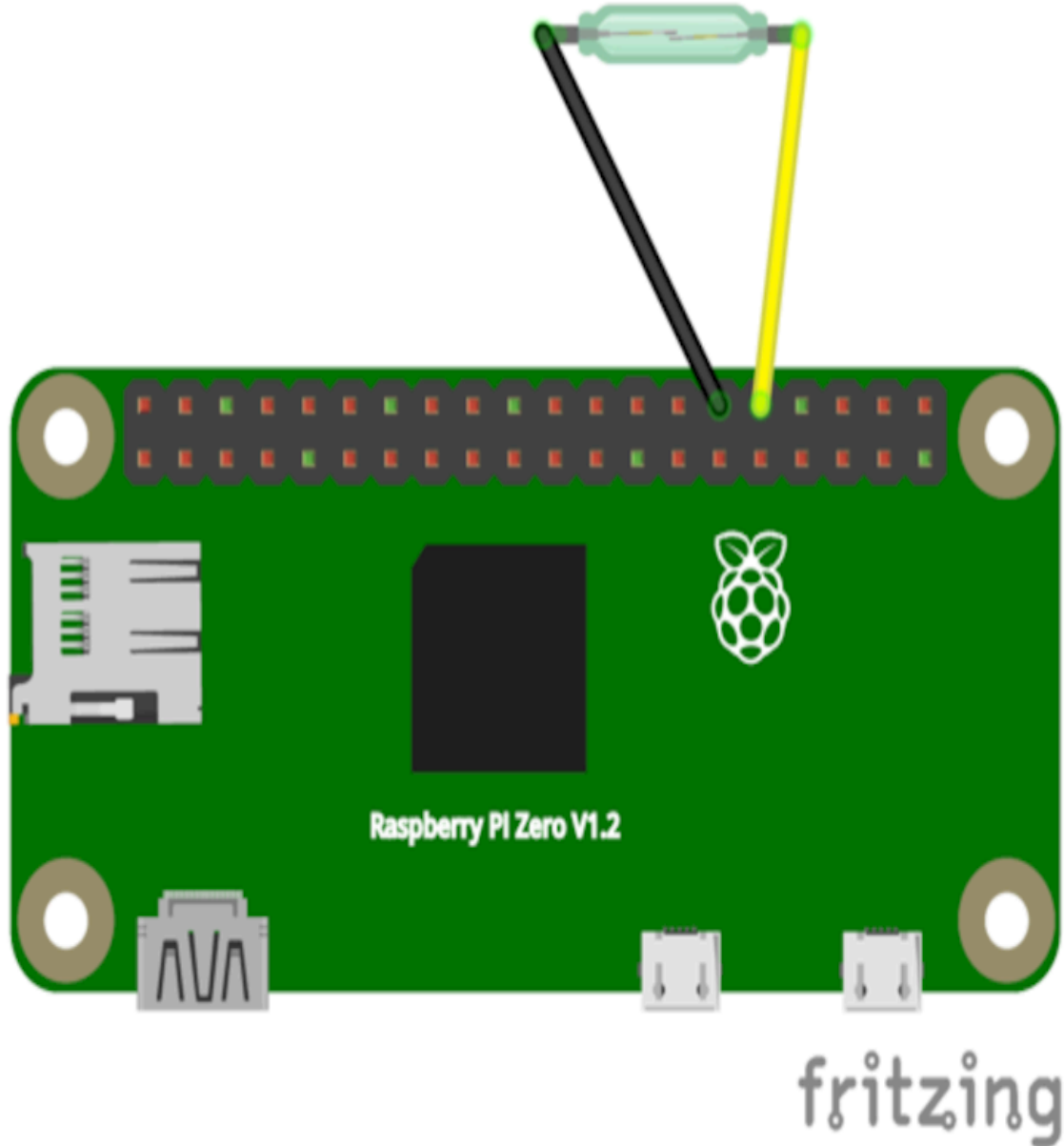
Wiring the Magnetic Switch to the GPIO

For this project, you need to wire the magnetic switch to the Raspberry Pi GPIO which allows you to detect when the switch state changes, then you need to attach the switch to your garage door, or any other door you want to use for this project. Let's start wiring the switch to the GPIO.

The magnetic contact switch comes in two parts: the first part is the actual switch that can be normally open or closed and the second part is a magnet. When the magnet approaches the switch, it changes its state. In a normally open switch, there's no connection between the two wires. When the magnet approaches, the switch closes and connects the wires. Conversely, in a normally closed switch, there's contact between the two wires by default, and the switch disconnects the wires when the magnet approaches.

You can use either a normally open or a normally closed switch for this project. For the example in the book, we're using a normally closed switch. In case you're using a normally open switch, reverse the order of "Open" and "Closed" parameters in the code.

The magnetic switch comes with two wires. You can connect those wires to two jumper cables to extend their lengths and make it easier to plug into the Pi Zero 2W pins. Then connect one of the wires to a GPIO pin and the other to a ground (GND) pin. You can use any of the GPIO pins, but for this example, we're using GPIO 12 pin which is the fifth from the right and the ground pin right next to it, like the following diagram:



With the switch wired to the Raspberry Pi, attach the switch side to the wall near the door, then attach the magnet part to the door, ensuring that they are close to each other when the door is closed. To ensure the switch can detect the magnet, they need to be at a maximum of 15 millimeters, or half an inch

from each other. This distance can vary with each device, so check your switch spec for details.

When the project is assembled, attached to the door, and the Pi Zero W powered up, it's time to write the code to check the door state.

Coding the Magnetic Switch

Before we dive into coding the entire project, let's focus on the core components and learn how to control the Raspberry Pi GPIO using Go code. This section provides the foundation to write software that interfaces with the GPIO. Upon completion, you'll understand how to initialize the GPIO, enable specific pins, set the pins as input or output, and enable the embedded Pull Up/Down resistor. These skills will enable you to develop this chapter's project and your own projects using the GPIO later on.

Later in this chapter, after you understand how to read the switch state using Go and the GPIO, you'll add the API and notification features to the project.

To control the GPIO, you'll use the external package `go-rpio`.^[56] Start by creating a directory for this example and initializing the Go module for it:

```
$ mkdir magnetic
$ cd magnetic
$ go mod init magnetic
go: creating new go.mod: module magnetic
```

Then, write the code into the `main.go` file. Start by importing the required packages:

[garagedoor/magnetic/main.go](#)

```
package main

import (
    "fmt"
    "os"

    "github.com/stianeikeland/go-rpio/v4"
)
```

Next, define a constant value for the pin number you used to connect the switch, in this case, pin 12:

[garagedoor/magnetic/main.go](#)

```
const pinNumber = 12
```

Then, add a new type `state` as an alias to the type `rpio.State` and attach a method named `String` to it to print the state as `Open` or `Closed` instead of 0 or 1:

[garagedoor/magnetic/main.go](#)

```
type state rpio.State
```



```
func (s state) String() string {  
    if s == state(rpio.Low) {  
        return "Open"  
    }  
  
    return "Closed"  
}
```

When you define a method named `String` that takes no inputs and returns a `string`, you're implicitly implementing Go's `fmt.Stringer` interface which automatically applies the defined format when you use the given type in a text context, like when printing the value.

Next, define the `main` function which opens and maps GPIO memory into the program, sets pin 12 as an input pin, and reads the pin state:

[garagedoor/magnetic/main.go](#)

```
func main() {  
    if err := rpio.Open(); err != nil {  
        fmt.Println(err)  
        os.Exit(1)  
    }  
}
```

```
// Unmap gpio memory when done
defer rpio.Close()

pin := rpio.Pin(pinNumber)

pin.Input()
rpio.PullMode(pin, rpio.PullUp)
door := state(pin.Read())

fmt.Println("Door is:", door)
}
```

Note that you're using the method `pin.Input()` to set the GPIO pin as an input pin, to receive the signal from the switch. You're also using the function `rpio.PullMode` with parameter `rpio.PullUp` to enable the Pull Up resistor for the given pin, ensuring the switch signal is stable.

Now, save the file and download the required packages by using the command `go mod tidy`:

```
$ go mod tidy
go: finding module for package
github.com/stianeikeland/go-rpio/v4
```

Then run the application to test it. You should see the switch state according to the magnet position. For example, if the door is closed:

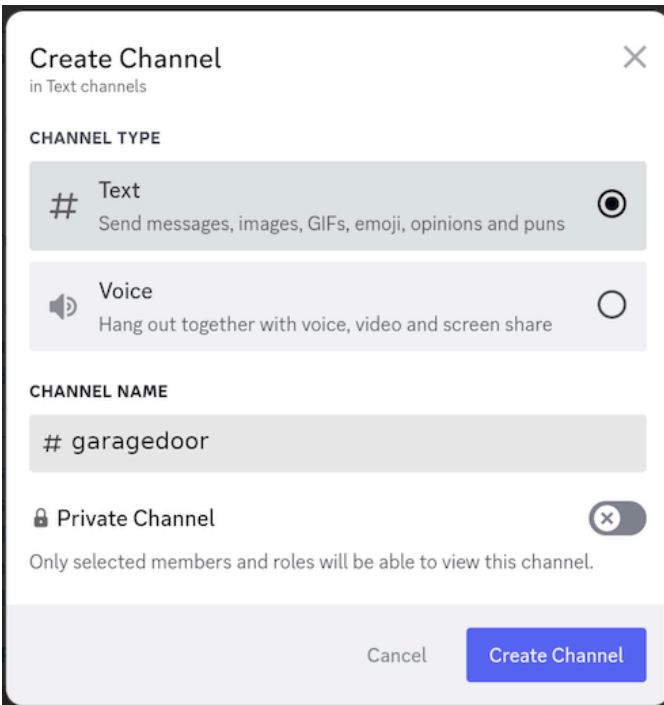
```
$ go run .
```

```
Door is: Closed
```

Now that you can control the GPIO via code, let's expand the example with additional features.

Sending Notifications

Discord is one of the largest chat services available today. The platform offers a very generous free tier with a rich API that we can use for our project needs. Once you set up and log into your Discord account, create a new channel on your server by selecting the + symbol to the right of the TEXT CHANNELS label. In the pop-up, select the default #Text channel type, and name the channel something apropos of the subject matter. In this case, call the channel [#garagedoor](#). We recommend using a private server for this case but if you share this server with someone else, you can also toggle on the Private Channel option so only you can view the [#garagedoor](#) channel on your Discord server. Your New Channel dialog should look similar to the one shown in the next image.

A screenshot of the 'Create Channel' dialog box in Discord. The dialog has a title bar with 'Create Channel' and a close button. Below the title, it says 'in Text channels'. The 'CHANNEL TYPE' section has two options: 'Text' (selected with a radio button) and 'Voice'. The 'Text' option has a description: 'Send messages, images, GIFs, emoji, opinions and puns'. The 'Voice' option has a description: 'Hang out together with voice, video and screen share'. The 'CHANNEL NAME' section has a text input field containing '# garagedoor'. Below the name field, there is a 'Private Channel' toggle switch, which is currently turned on. A note below the toggle says 'Only selected members and roles will be able to view this channel.' At the bottom, there are two buttons: 'Cancel' and 'Create Channel'.

Select the [Create Channel](#) button to continue. If you selected the Private Channel option, select [Next](#) to continue to the next screen to list groups and any subscribers to your server you would like to grant access to the new [#garagedoor](#) channel. Select those groups or individuals you want to allow to get notified from their Discord client anytime a new message is posted to the channel.

With the new [#garagedoor](#) channel created and displayed in the TEXT CHANNELS listing, select the sprocket icon to the right of the [#garagedoor](#) label to edit the channel. From this dialog box, select the Integrations menu item, followed by clicking on the [New Webhook](#) button. Give the webhook a name like [Garage Door](#)

[Webhook](#) and then select the [Save Changes](#) button to create the webhook.

Once you have saved your changes, select the [Copy Webhook URL](#) button to copy the newly created webhook URL. This URL will be pasted as an environment variable in our Go program as the address to post new notifications. Using an environment variable makes it easy to modify the URL string without recompiling and redeploying the code. This can happen if the URL is compromised, or the target needs to change to a different text channel.

Another benefit of not including such sensitive, confidential strings like webhook URLs in your source code is that you don't accidentally disclose the URL in public source management services like GitHub. If you are using the Gitea source code repository that was set up in [*Managing Source Code with Gitea*](#), and are the only user on that system, and if you can only log into the Pi running Gitea on your local network, the exposure risk is not as great. But you should still practice safe data management even in those situations. Hence, we'll expose the [Garage Door Webhook](#) URL only as an environment variable within the final application container for now.

With the active webhook URL ready for use, we now have all the pieces in place to complete the Go code for this project, so let's build the code listing.

Writing the Software

With the basic functionality in place, it's time to improve the program to be more useful. You'll make the program more robust by adding a configuration file, expose the door state using a HTTP API, and send a Discord notification in case you forgot you left the door open at night.

Start by creating a directory for this new version, and initializing the Go module:

```
$ mkdir final
$ cd final
$ go mod init doorcheck
go: creating new go.mod: module doorcheck
```

Now add the code to read a configuration file in [yaml](#) format. Use the external package `yaml.v3`^[57] for this. Add the code to file `config.go`:

[garagedoor/final/config.go](#)

```
package main

import (
    "errors"
    "log"
```



```
    "os"
    "time"

    "gopkg.in/yaml.v3"
)

type yamlHour struct {
    t time.Time
}

func (yh *yamlHour) UnmarshalYAML(v *yaml.Node)
error {
    if v.Kind != yaml.ScalarNode {
        return errors.New("value is not scalar")
    }

    var err error

    yh.t, err = time.Parse("3:04pm", v.Value)

    return err
}

type config struct {
```

```

    SwitchPinNumber int
`yaml:"switch_pin_number"`

    NightStart      yamlHour `yaml:"night_start"`
    NightEnd        yamlHour `yaml:"night_end"`
}

func newConfig(configFile string) (*config, error)
{
    cf, err := os.Open(configFile)
    if err != nil {
        return nil, err
    }

    defer cf.Close()

    var cfg *config
    if err := yaml.NewDecoder(cf).Decode(&cfg);
err != nil {
        return nil, err
    }

    if cfg.SwitchPinNumber == 0 {
        log.Println(cfg)
    }
}

```

```
        return nil, errors.New("switch pin needs  
to be defined")  
    }  
  
    if cfg.NightStart.t.IsZero() {  
        var err error  
        cfg.NightStart.t, err = time.Parse(  
"3:04pm", "9:00pm")  
        if err != nil {  
            return nil, err  
        }  
    }  
  
    if cfg.NightEnd.t.IsZero() {  
        var err error  
        cfg.NightEnd.t, err = time.Parse("3:04pm",  
"7:00am")  
        if err != nil {  
            return nil, err  
        }  
    }  
  
    return cfg, nil  
}
```

This part of the code defines a new `struct` type named `config` that reads three configuration values from the configuration file:

- `switch_pin_number`: which allows the user to change the GPIO pin number used for the switch
- `night_start`: which represents the time the nightly notification period starts
- `night_end`: which represents the time the nightly notification period ends

Save and close `config.go` and create `main.go` by adding the `import` section with the required dependencies:

[garagedoor/final/main.go](#)

```
package main

import (
    "bytes"
    "encoding/json"
    "flag"
    "fmt"
    "log"
    "net/http"
    "net/url"
    "os"
```

```
    "time"  
  
    "github.com/stianeikeland/go-rpio/v4"  
)
```

Then, define the same type `state` with the method `String` to print `Open` or `Closed` like you did in [Coding the Magnetic Switch](#):

[garagedoor/final/main.go](#)

```
type state rpio.State  
  
func (s state) String() string {  
    if s == state(rpio.Low) {  
        return "Open"  
    }  
  
    return "Closed"  
}
```

Now, add a function `setupGPIO` to initialize the GPIO:

[garagedoor/final/main.go](#)

```
func setupGPIO(pinNumber int) (rpio.Pin, error) {  
    if err := rpio.Open(); err != nil {  
        log.Println("Error opening GPIO:", err)  
    }  
}
```

```
        return 0, err
    }

    pin := rpio.Pin(pinNumber)

    pin.Input()
    rpio.PullMode(pin, rpio.PullUp)

    return pin, nil
}
```

Then, add the function `getDoorState` to read the switch state from the GPIO:

[garagedoor/final/main.go](#)

```
func getDoorState(pin rpio.Pin) state {
    return state(pin.Read())
}
```

These two functions have the same functionality from the previous initial example but split into two smaller functions to make it easier to reuse them in the `main` function.

Next, add the function `isNight` to verify if the door is open at night:

[garagedoor/final/main.go](#)

```
func isNight(start, end time.Time) bool {
    cur := time.Now().Format("15:04")

    now, err := time.Parse("15:04", cur)
    if err != nil {
        log.Println(err)
        return false
    }

    if end.Before(start) {
        end = end.Add(24 * time.Hour)
    }

    if now.Before(start) {
        now = now.Add(24 * time.Hour)
    }

    return now.After(start) && now.Before(end)
}
```

Then, add function `sendNotification` to send a Discord notification if the door is open at night:

[garagedoor/final/main.go](#)

```

func sendNotification(discordWebhook, message
string) {
    u, err := url.Parse(discordWebhook)
    if err != nil {
        log.Println("Invalid Discord webhook URL:"
, err)

        return
    }

    v := url.Values{}
    v.Set("wait", "true")
    u.RawQuery = v.Encode()

    payload := struct {
        Content string `json:"content"`
    }{

```

```

        Content: message,
    }

    var body bytes.Buffer
    if err :=
json.NewEncoder(&body).Encode(payload); err != nil
{

```



```
        log.Println("Error creating JSON payload:"
, err)

        return
    }

    request, err :=
http.NewRequest(http.MethodPost, u.String(),
&body)

    if err != nil {
        log.Println("Error creating Discord
request:", err)
        return
    }

    request.Header.Add("Content-Type",
"application/json")

    client := http.Client{
        Timeout: 10 * time.Second,
    }

    response, err := client.Do(request)
    if err != nil {
        log.Println("Error sending Discord
request:", err)
```

```

        return
    }

    defer response.Body.Close()

    if response.StatusCode != http.StatusOK {
        log.Printf(
            "Invalid response from Discord
channel: %s",
            response.Status,
        )
    }
}

```

Now, complete the nightly door check functionality by adding the function `checkDoor` which constantly checks the door state and sends a notification if it's open at night. Change the notification frequency time in the `for range` loop if one minute is too short:

[garagedoor/final/main.go](#)

```

func checkDoor(pin gpio.Pin, cfg *config,
discordWebhookURL string) {
    for range time.Tick(1 * time.Minute) {
        doorState := getDoorState(pin)
    }
}

```

```
        log.Println("Door state:", doorState)
        if doorState == state(rpio.Low) {
            if isNight(cfg.NightStart.t,
cfg.NightEnd.t) {
                message := fmt.Sprintf(
                    "Door open at night:",
time.Now(),
                )
```

```
            log.Println(message)
            go sendNotification(
                discordWebhookURL,
                message,
            )
        }
    }
}
```

Next, implement the HTTP API server by defining an API handler function to handle requests to obtain the door state using path `/getdoor`:

[garagedoor/final/main.go](#)

```

func doorStateHandler(pin rpio.Pin)
http.HandlerFunc {
    return func(w http.ResponseWriter, _
*http.Request) {
        doorState := getDoorState(pin)
        log.Println("Door state:", doorState)

        response := struct {
            DoorState      state
`json:"door_state"`
            DoorStateText string
`json:"door_state_text"`
        }{
            DoorState:      doorState,
            DoorStateText: fmt.Sprintf(doorState),
        }

        w.Header().Set("Content-Type",
"application/json")

        if err :=
json.NewEncoder(w).Encode(response); err != nil {
            log.Println("Error replying door
state:", err)
        }
    }
}

```

```
}  
  
}
```

Then, complete the HTTP API server by adding the multiplexer function which attaches all the routes and defines a root route:

[garagedoor/final/main.go](#)

```
func newMux(pin rpio.Pin) http.Handler {  
    mux := http.NewServeMux()  
  
    mux.HandleFunc("GET /", func(w  
http.ResponseWriter, _ *http.Request) {  
        fmt.Fprintln(w, "Door status API  
running...")  
    })  
  
    mux.Handle("/getdoor", doorStateHandler(pin))  
  
    return mux  
}
```

Finally, put it all together by adding the **main** function as the program entry point. This function reads the configuration file and Discord webhook URL from an environment file; initializes

the GPIO, then starts a goroutine to constantly check the door state; and finally, starts the HTTP API server:

[garagedoor/final/main.go](#)

```
func main() {  
    c := flag.String("c", "config.yml", "Config  
file")  
    flag.Parse()  
  
    cfg, err := newConfig(*c)  
    if err != nil {  
        log.Println("Error opening config file:",  
err)  
        os.Exit(1)  
    }  
  
    discordWebhookURL, ok := os.LookupEnv(  
"DISCORD_WEBHOOK_URL")  
    if !ok {  
        log.Println("'DISCORD_WEBHOOK_URL' env var  
is required")  
        os.Exit(1)  
    }  
  
    pin, err := setupGPIO(cfg.SwitchPinNumber)
```

```

if err != nil {
    log.Println("Error opening GPIO:", err)
    os.Exit(1)
}

defer rpio.Close()

go checkDoor(pin, cfg, discordWebhookURL)

s := &http.Server{
    Addr:           ":3060",
    Handler:        newMux(pin),
    WriteTimeout: 10 * time.Second,
}

log.Println("Starting API server on port :3060")

if err := s.ListenAndServe(); err != nil {
    log.Println(err)
    os.Exit(1)
}
}

```

With the code in place, save the `main.go` file and download the required dependencies with `go mod tidy`:

```
$ go mod tidy
```

Then test the program by running `go run .`:

```
$ go run .  
2024/01/31 00:03:37 Error opening config file:  
open config.yml:  
    no such file or directory  
exit status 1
```

The program fails to execute because the configuration file is required. Let's configure the program next.

Configuring and Testing the Application

To allow this program to work, you need to configure it by providing two requirements:

1. A configuration file, `config.yml`, which configured the pin number to use as well as the notification period
2. The Discord webhook URL using environment variable `DISCORD_WEBHOOK_URL`

Start by adding a configuration file with the desired parameters according to your requirements. For example:

[garagedoor/final/config.yml](#)

```
---  
  
switch_pin_number: 12  
night_start: "10:30pm"  
night_end: "7:00am"
```

Next, you need to provide Discord's webhook URL that you created in [Sending Notifications](#), to send Discord notifications. Then export the variable:

```
export DISCORD_WEBHOOK_URL=  
<your_discord_webhook_URL>
```

Now you can finally run the program and test it:

```
$ go run .  
2024/01/31 00:17:37 Starting API server on port  
:3060  
2024/01/31 00:18:37 Door state: Closed
```

You can query the door state at any time by sending a HTTP GET request to your Pi Zero 2W IP address or hostname, on port 3060:

```
$ curl -s http://<PI_ZERO_2W_IP>:3060/getdoor  
{  
  "door_state": 1,  
  "door_state_text": "Closed"  
}
```

If the door is open during the notification period set in your configuration file, and your Discord webhook is correctly configured, you'll receive a Discord notification similar to: **Door open at night:2024-01-30 01:49:19.934671452 -0500 EST m=+60.056133938.**

As the final step, let's containerize the application to run it with Docker.

Containerizing the Deployment

To facilitate running this application as a system service, isolated from other processes in the same Raspberry Pi Zero 2W, let's create a container image for it. By running an application as container, you can also set it to automatically start when the Pi Zero 2W boots up or in case the application fails for any reason.

Start by adding a [Dockerfile](#) with the image-building instructions. Like other examples in the book, we're using the staged approach where we build the application using the official Go image, then copy the application to another image running Alpine Linux:

[garagedoor/final/Dockerfile](#)

```
FROM docker.io/golang:1.22 AS builder
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm go build \
    -ldflags="-s -w" \
    -o doorcheck \
    .
```

```
FROM docker.io/alpine:latest
RUN mkdir /app && adduser -h /app -D doorcheck
WORKDIR /app
COPY --chown=doorcheck --from=builder
/app/doorcheck .
ENTRYPOINT ["/app/doorcheck"]
```

Note that, unlike other images you built for Raspberry Pi 4 or 5, you're building an image for the Pi Zero 2W which uses a 32-bit ARM CPU. Therefore, the Go build parameter `GOARCH` is set to `arm` instead of `arm64`.

Now, build the image using `docker build`:

```
$ docker build -t doorcheck:v1 .
```

Check the image created:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
doorcheck	v1	229f723120e7	7 minutes ago
SIZE	10.5MB		

Now, launch the container using `docker run`. Use the option `-e` to expose the environment variable `DISCORD_WEBHOOK_URL` to the

container. Use option `-v` to mount the config file `config.yml` from your host machine into the `/etc/config.yml` file in the container. Mount the file `/etc/localtime` from the host to the container so the container runs in the same time zone as the host, avoiding time conversions. Expose port 3060 for the API by using option `-p`. Use option `--device` to expose the `/dev/gpiomem` and `/dev/mem` devices, which allows the container access to the GPIO. Finally, append the application parameter `-c /etc/config.yml`, allowing the application to read the configuration file mounted from your local machine:

```
$ docker run -d \
    --name doorcheck \
    --restart=always \
    -v $(pwd)/config.yml:/etc/config.yml \
    -v /etc/localtime:/etc/localtime \
    -p 3060:3060 \
    -e DISCORD_WEBHOOK_URL=${DISCORD_WEBHOOK_URL} \
    --device /dev/gpiomem --device /dev/mem \
    doorcheck:v1 \
    -c /etc/config.yml
```

Ensure the container is running by using `docker ps`:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
441922a6339c	doorcheck:v1	"/app/doorcheck -c /..."
2 minutes ago		
Up 2 minutes	0.0.0.0:3060->3060/tcp	
doorcheck		

You can also check the application logs:

```
$ docker logs -f doorcheck
```

```
2024/01/31 01:08:34 Starting API server on port  
:3060  
2024/01/31 01:08:53 Door state: Open
```

Finally, test the application by issuing a GET request to the API server:

```
$ curl -s http://192.168.10.131:3060/getdoor  
{  
  "door_state": 0,  
  "door_state_text": "Open"  
}
```

Now you can rest assured you'll never forget to close the garage door at night.

Next Steps

This project provides a useful system to verify if your garage door is open or closed and notify you in case you left it open at night. In addition, this project presents the basics you need to use the Raspberry Pi GPIO. You can build on this knowledge to use other types of sensors, like motion sensors or distance sensors, to build even more complex and smart solutions for your home automation. In fact, in Chapter 7, [*Watching the Birds*](#), you'll use the GPIO again with a PIR motion sensor to snap nice wildlife pictures.

You can also expand this project by using the magnetic switch in other places like the front or back door, and any other place you need a quick way to check if two parts are connected. On the software side, you can improve the project by adding new features such as avoiding continuous notifications, or sending a notification if the door has been open for more than a few hours during the day. There are many possibilities, which make building your own projects fun.

In the next chapter, you'll use the Hue colored light strip to build an automated and visual way to monitor the weather outside. See you there.

FOOTNOTES

[55]<https://projects.raspberrypi.org/en/projects/physical-computing/1>

[56]<https://github.com/stianeikeland/go-rpio/v4>

[57]<https://gopkg.in/yaml.v3>

Chapter 6

Lighting the Weather

In Chapter 4, [*Networking a Temperature Monitor*](#), we created a network-enabled temperature probe to report current temperature conditions. While we could use that same approach for monitoring outdoor temperatures as well, the Pico W would need to be shielded in a weather-resistant case. If the external temperatures became extreme, the Pico W might permanently fail. Fortunately, enough external temperature sensors are already monitoring outdoor weather conditions. We can simply poll available APIs for JSON payloads containing current weather condition values. But rather than simply report the number, we can use colored bulb lighting to visually indicate whether it's cold, comfortable, or hot outside. We'll send color commands to a Philips Hue lighting setup based on the temperature values received by the outdoor temperature API call.

Project's Hardware Requirements

This project requires these components:

- *Raspberry Pi server*: A Raspberry Pi 3, 4, or Zero 2 to act as the application server.
- *Hue base station*: Part of the Hue Start Kit, it maintains the inventory and state of the Hue lighting in your home.
- *Hue multi-colored lighting strip*: The light we'll program based on the outdoor temperature.

For more details, consult [*Adding Other Hardware Components*](#).

After you complete this project, it's going to look like the [picture](#), where you can see the light with two different states, blue and red.



If the light is blue, grab a jacket because it's cold outside. Is the light red? Then it's warm enough to enjoy the outdoors wearing a T-shirt.

In this chapter, you'll use REST API calls to obtain the external weather information and control the lights. Handling APIs is an important skill to learn as it allows you to obtain information from a large pool of sources. It also allows you to control automation devices that expose such API interfaces, expanding the range of home automation controllers you can use on your own projects.

Let's get started.

Polling the Weather

To query the current weather conditions, we need access to an API that can provide those details. Fortunately, a service called OpenWeather^[58] offers a free tier for developers that allows a copious number of calls to their service. Sign up^[59] to request a free API key. You'll need this key when making calls to OpenWeather's API. In particular, it'll be used to poll the current outdoor temperature in your area.

Once you have your free OpenWeather API key, test it out by polling the current temperature in your area with this small Go program, which uses the `openweathermap` package to query the OpenWeather API. Replace the `API_KEY` and `ZIP_CODE` values with your own before running the test. If you live outside the United States, replace the country code as well:

```
package main

import (
    "fmt"
    "log"
    "os"

    owm "github.com/briandowns/openweathermap"
```

```
)

func main() {
    w, err := owm.NewCurrent("F", "EN", API_KEY)
    if err != nil {
        log.Fatalln(err)
    }

    w.CurrentByZip(ZIP_CODE, "US")
    fmt.Println(w.Main.Temp)
}
```

Save the file as `openweathertest.go` and run `go mod tidy` to download GitHub user Brian Downs's `openweathermap` Go library. This library makes it very easy to use OpenWeather's API in the Go language environment.

With everything nice and tidy, run the program using the usual Go run syntax to test your API key, like this:

```
$ go run openweathertest.go
```

Assuming the values you replaced for your `API_KEY` and `ZIP_CODE` are valid, you should see the current temperature output in Fahrenheit. If you prefer the temperature scale to be

reported in Celsius, change the parameter in `NewCurrent` to `C`, like this:

```
w, err := owm.NewCurrent("C", "EN", API_KEY)
```

Congratulations! You're now able to poll the current outdoor temperature in your area. The OpenWeather API offers many other options you can explore. The free tier is somewhat limited in the level of detail and forecast information it provides, but enough data is available to be useful for our project. Feel free to experiment with other calls to the API, as well as poll other geographic regions where you might be interested in the current temperature.

In the next section, we'll use the current temperature value received from the `w.Main.Temp` variable and light a Hue Philips color lightstrip to visually reflect the current outdoor temperature.

Changing the Color

The Hue base station maintains the inventory and state of the Hue lighting in your home, and the lighting strip is what we'll program based on the outdoor temperature. In order to make the color of the lighting meaningful, we need to determine temperature ranges with which to display the appropriate color. For example, the color blue is frequently associated with cold. So setting the light strip to that color anytime the temperature is below 50 degrees Fahrenheit would indicate cooler temperatures outside. Conversely, the color red typically indicates hot. Thus, anytime the outdoor temperature is hotter than 90 degrees, change the light strip color to red. Here are the color recommendations between those two values:

```
Blue    = Below 50 degrees
Yellow  = Between 51 and 65 degrees
Green   = Between 66 and 79 degrees
Orange  = Between 80 and 89 degrees
Red     = Above 90 degrees
```

Let's codify those rules in Go using a `switch` statement and append it to the `openweathertest.go` program to test. To make it easier to code the `switch` statement, first assign the temperature

returned by the API call `w.Main.Temp` to a new variable `currentTemp` in the `main` function:

```
var currentTemp = w.Main.Temp
```

Then, append this `switch` block to the end of the `main` function to display the color:

```
switch {
case currentTemp < 51:
    fmt.Println("Blue")
case currentTemp >= 51 && currentTemp < 66:
    fmt.Println("Yellow")
case currentTemp >= 66 && currentTemp < 80:
    fmt.Println("Green")
case currentTemp >= 80 && currentTemp < 90:
    fmt.Println("Orange")
case currentTemp >= 90:
    fmt.Println("Red")
}
```

Run the program via the usual `go run openweathertest.go` command, and depending on the current outdoor temperature, the appropriate color should display right after the actual temperature value that was evaluated. Now that the proper

respective color is indicated based on the outside temperature, it's time to hook up and connect to the Hue base station.

Programming the Hue

Before we can start programming the Hue from a Go application, make sure that you have correctly set up the Hue base station on your network, and added the light strip to the Hue's inventory. You can use the official Philips Hue app, available from either the Android^[60] or iOS^[61] app stores.

Once you can remotely control your Hue light strip from the Hue app, you are ready to configure a new user account on the Hue base station. You'll use this account to interact with and send commands from your Go application.

Several Hue libraries for Go are available on GitHub. The one that works best with this particular project was created by GitHub user Collinux, called `gohue`. This rudimentary library makes it easy to connect, control, and set basic colors on Hue lighting.

Before we can remotely control Hue-managed lights, we need an authorized User ID to log into the Hue base station. The `gohue` library provides a `CreateUser` function that instructs the Hue to

generate a new User ID for this purpose. To do so, write the following Go program:

```
package main

import (
    "github.com/collinux/gohue"
)

func main() {
    bridgesOnNetwork, _ := hue.FindBridges()
    bridge := bridgesOnNetwork[0]
    username, _ := bridge.CreateUser("gohomeuser")
    fmt.Println(username)
}
```

Save the code as `createhueuser.go` and run it with the Hue base station nearby. You'll need to press the large button on the top of the Hue base station to authorize the User ID creation request when the `createhueuser.go` program is run.

```
$ go run createhueuser.go
```

Remember to copy the username ID that is generated after authorizing the request on the Hue. You'll use this ID for

programmatic Hue base station access.

Now that you have created the new authorized Hue account, you can use it when programmatically manipulating your Hue lights. Verify that this newly generated User ID allows you to control your Hue light by creating a simple program that turns the light on. The following sample code assumes you named your light “Desk” using the Hue smartphone app. You can reference this light in your code using the function [GetLightByName\(\)](#) and whatever actual name you assigned to the light using the Hue smartphone app.

```
package main

import (
    "github.com/collinux/gohue"
)

func main() {
    HUE_ID := os.Getenv("HUE_ID")
    HUE_IP_ADDRESS := os.Getenv("HUE_IP_ADDRESS")

    bridge, _ := hue.NewBridge(HUE_IP_ADDRESS)

    bridge.Login(HUE_ID)
```

```
deskLight, _ := bridge.GetLightByName("Desk")

deskLight.On()

}
```

Save the code as `huetest.go` and make sure your `HUE_ID` and `HUE_IP_ADDRESS` environment variables are properly assigned. Then run the code via the typical `go run` command.

```
$ go run huetest.go
```

If everything runs successfully, your targeted Hue light should turn on. Now that we can control lights from Go, let's expand our code to run as a service.

Putting It All Together

Now that you have the individual pieces in place to check the weather and control the Hue lights, let's put it all together by expanding the program to check the temperature status at any time from any other application that sends a request to the service. In addition, let's also allow the results to be exported regularly to a Prometheus exporter that can be visualized in Grafana. Lastly, we'll want to remove the statically defined values in our code and move it into a configuration file that can be easily edited. This way, if there are changes to the color scheme, API keys, locations, or other variables, we don't have to edit and recompile the source code to activate those changes.

Start the final version by creating the HTML file template for the home page with a link to redirect the user to the metrics endpoint. Later, you'll use Go's `embed` package to embed this file into the final binary, like you did in [*Creating the Prometheus Exporter*](#):

[`lightingweather/rootPage.html`](#)

```
<html>
  <head>
    <title>External Weather Temperature
    Exporter</title>
```

```
</head>
<body>
    <h1>External Weather Temperature
Exporter</h1>
    <p><a href= '/metrics'>metrics</a></p>
</body>
</html>
```

Next, create the file `lightingweather.go` for your program, and add the `package` definition:

[`lightingweather/lightingweather.go`](#)

```
package main
```

As the program's functionality expands, so too are the external libraries needed to assist with these new capabilities. Add the import statement containing all the libraries needed to get the program to run successfully:

[`lightingweather/lightingweather.go`](#)

```
import (
    _ "embed"
    "flag"
    "log"
    "math"
```

```
    "net/http"
    "os"
    "time"

    "github.com/prometheus/client_golang/prometheus"

    "github.com/prometheus/client_golang/prometheus/promhttp"

    "github.com/prometheus/client_golang/prometheus/promauto"

    owm "github.com/briandowns/openweathermap"
    hue "github.com/collinux/gohue"
)
```

Note that several additional standard Go libraries such as [net/http](#) and [time](#) were added, along with the Prometheus Go libraries we used in the previous project.

Next, we'll create a variable for the Prometheus exporter home page by embedding the [rootPage.html](#) file using the [embed](#) package. While not absolutely required, it's helpful to have this descriptive web page for exporters to document their purpose, along with any other important instructions regarding the metrics being collected.

[lightningweather/lightningweather.go](https://github.com/lightningweather/lightningweather.go)

```
//go:embed rootPage.html  
var rootPageHTML []byte
```

The biggest change we'll make to the prototype code we wrote earlier is to encapsulate getting the temperature and changing the lighting into a single function. Additionally, we'll want to incorporate the values assigned in the configuration file to avoid hardcoding these values into the function. Pay close attention to the **for** statement within this function that determines whether to execute the function immediately, or wait 30 minutes to do so. This way, the application can respond to both immediate requests from a different application, or to the Prometheus exporter that will provide the current temperature value every 30 minutes.

[lightningweather/lightningweather.go](https://github.com/lightningweather/lightningweather.go)

```
func lightweather(cfg *config, chRefresh <-chan  
struct{}) {  
    externalWeatherTemp :=  
promauto.NewGauge(prometheus.GaugeOpts{  
        Name: "external_weather_temperature",  
    })  
  
    run := func() {
```

```
        log.Println("INFO: Getting current temperature")

        currentTemp, err :=
getCurrentTemperature(cfg)
        if err != nil {
            log.Println("ERROR:", err)
        }

        externalWeatherTemp.Set(float64
(currentTemp))

        log.Println("INFO: Setting light")
        if err := setLight(cfg, currentTemp); err
!= nil {
            log.Println("ERROR:", err)
        }
    }

    for {
        select {
        case <-chRefresh:
            run()
        case <-time.Tick(30 * time.Minute):
            run()
        }
```

```
    }  
  }  
}
```

Next, in order to provide scalability to the application, we'll use a really helpful feature in Go called **channels** and **goroutines** to run the **lightweather** function concurrently. We're using the function **make** to create a buffered channel, allowing caching of requests. This way, if Prometheus is polling the application's values at the exact same time as another external application is doing so, these requests will be processed in the order they were received rather than being discarded.

In Go, you can use a **channel** to pass values across different **goroutines** running concurrently. In this case, we're using a channel of type **empty struct** - **struct{}** since we don't need to pass any real values, but rather, just to signal a refresh. By using empty **struct**, we avoid any memory allocations. Add this code to your program:

[lightingweather/lightingweather.go](#)

```
func newMux(cfg *config) http.Handler {  
    mux := http.NewServeMux()
```

```
    mux.HandleFunc("/", func(w
http.ResponseWriter, _ *http.Request) {
    log.Println("INFO: Received request root")
    w.Write(rootPageHTML)
})

chRefresh := make(chan struct{}, 2)

go lightweight(cfg, chRefresh)

chRefresh <- struct{}{}

mux.HandleFunc("POST /refresh",
    func(w http.ResponseWriter, _
*http.Request) {
    log.Println("INFO: Received refresh
request")

    chRefresh <- struct{}{}
    w.WriteHeader(http.StatusAccepted)
    w.Write([]byte("Refresh request
accepted"))
})
```

```
    mux.Handle("/metrics", promhttp.Handler())

    return mux
}
```

Pay attention to how we modified the `newMux` function from the previous project to add a channel and two routes to handle the two types of requests discussed earlier. We send the `empty struct` value over the channel by using the syntax `struct{}`, which defines an anonymous value `{}` of type `struct{}`.

In the program's `main` function, we'll import the key values from the `config.yml` file, then start the HTTP server, listening on port 3040. If any errors occur while attempting to start the server, those errors will be logged to the standard out console.

[lightingweather/lightingweather.go](https://lightingweather.com/lightingweather.go)

```
func main() {
    c := flag.String("c", "config.yml", "Config
file")
    flag.Parse()

    cfg, err := newConfig(*c)
    if err != nil {
        log.Println("ERROR:", err)
    }
}
```

```

        os.Exit(1)
    }

    s := &http.Server{
        Addr:           ":3040",
        Handler:        newMux(cfg),
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
    }

    if err := s.ListenAndServe(); err != nil {
        log.Println("ERROR:", err)
        os.Exit(1)
    }
}

```

We're almost done. In this `main` package, we just have to define the `getCurrentTemperature()` and `setLight()` functions we called from the `lightweather()` function we created earlier.

[lightingweather/lightingweather.go](#)

```

func getCurrentTemperature(cfg *config) (int,
error) {

```

```
    w, err := owm.NewCurrent(cfg.Unit, cfg.Lang,
cfg.OWMAPIKey)

    if err != nil {
        return 0, err
    }

    err = w.CurrentByName(cfg.Location)

    return int(math.Round(w.Main.Temp)), err
}
```

```
func setLight(cfg *config, currentTemp int) error
{
    bridge, err := hue.NewBridge(cfg.HueIPAddress)
    if err != nil {
        return err
    }

    // hue-id, _ := bridge.CreateUser("create-new-
user")

    if err := bridge.Login(cfg.HueID); err != nil
{
        return err
    }
}
```

```
    weatherLight, err :=
bridge.GetLightByName(cfg.LightName)
    if err != nil {
        return err
    }

    if err := weatherLight.SetColor(
        pickColor(cfg, currentTemp)); err != nil {
        return err
    }

    return weatherLight.On()
}
```

Finally, save the file [lightingweather.go](#) with the main part of the program ready. Next, let's add a configuration feature that allows you to provide a configuration file to your application.

Configuring the Application Settings

To make your program more flexible, let's add a feature that allows you to specify required parameters using a configuration file in YAML format. This way, if the conditions change, you can update the configuration files without recompiling your application. Create a file `config.go` that's responsible for importing the key values stored in the `config.yml` file, like this:

[lightingweather/config.go](#)

```
package main

import (
    "errors"
    "fmt"
    "os"
    "sort"

    hue "github.com/collinux/gohue"
    "gopkg.in/yaml.v3"
)

var errInvalidColor = errors.New("invalid color")

type color struct {
```

```
Color      string `yaml:"color"`  
Threshold int     `yaml:"threshold"`  
}
```

```
var colorTranslate = map[string]*[2]float32{  
    "blue":    hue.BLUE,  
    "cyan":    hue.CYAN,  
    "green":   hue.GREEN,  
    "orange":  hue.ORANGE,  
    "pink":    hue.PINK,
```

```
    "purple":  hue.PURPLE,  
    "red":     hue.RED,  
    "white":   hue.WHITE,  
    "yellow":  hue.YELLOW,  
}
```

```
type config struct {  
    Unit      string `yaml:"unit"`  
    Lang      string `yaml:"lang"`  
    Location  string `yaml:"location"`  
    HueID     string `yaml:"hue_id"`  
    HueIPAddress string `yaml:"hue_ip_address"`  
    OWMAPIKey string `yaml:"owm_api_key"`
```

```

    LightName      string  `yaml:"light_name"`
    MaxColor       string  `yaml:"max_color"`
    Colors         []color  `yaml:"colors"`
}

func (cfg *config) sortColorRange() *config {
    sort.Slice(cfg.Colors, func(i, j int) bool {
        return cfg.Colors[i].Threshold <
cfg.Colors[j].Threshold
    })

    return cfg
}

func newConfig(configFile string) (*config, error)
{
    cf, err := os.Open(configFile)
    if err != nil {
        return nil, err
    }

    defer cf.Close()

    var cfg config

```

```
    if err := yaml.NewDecoder(cf).Decode(&cfg);
err != nil {
    return nil, err
}

for _, cl := range cfg.Colors {
    if _, ok := colorTranslate[cl.Color]; !ok
{
        return nil, fmt.Errorf("%w: %s",
            errInvalidColor, cl.Color)
    }
}

// Allow user to override OWM API Key with env
var
    if owmKey, ok := os.LookupEnv("OWM_API_KEY");
ok {
        cfg.OWMAPIKey = owmKey
    }

    return cfg.sortColorRange(), nil
}
```

```

func pickColor(cfg *config, curTemp int) *[2]
float32 {
    for _, cl := range cfg.Colors {
        if curTemp < cl.Threshold {
            return colorTranslate[cl.Color]
        }
    }

    return colorTranslate[cfg.MaxColor]
}

```

This part of the program defines a `struct` named `config` to store important configuration options such as your `Location`, `API Key`, and color thresholds, among others. Then it defines a function `newConfig` that reads that information from a YAML configuration file using the package `gopkg.in/yaml.v3`. Since your API key could be sensitive information, this function also allows overriding it using an environment variable `OWM_API_KEY` so you can provide the key at run time instead of saving it to a file that could end up in a Git repository.

Finally, this is a sample `config.yml` file where you'll replace the placeholder values with values relevant to you, such as:

- the API key you obtained from OpenWeather

- whether to report temperature values in Celsius (C) or Fahrenheit (F)
- which language and location you prefer to query
- the Hue base information like Hue ID and IP address
- multi-color light name you want to use
- the color thresholds to display on the light, based on unit and values

[lightingweather/config.yml](#)

```
---
unit: "C"
lang: "EN"
location: "Chicago, US"
hue_id: "12345"
hue_ip_address: "192.168.0.33"
owm_api_key: "123"
light_name: "Desk"

max_color: red

colors:
  - color: orange
    threshold: 30
  - color: green
    threshold: 20
```

```
- color: yellow
  threshold: 25
- color: blue
  threshold: 5
```

Again, it's important to note just how helpful and efficient it is to store these variables in a configuration file, so they can be easily modified without having to recode and recompile your project every time your lighting or location preferences change.

Once these files are in place, containing valid OpenWeather values and Hue lighting variables, you can run and test the project via the usual `go run` command.

```
$ go run lightingweather.go config.go
2023/07/29 20:54:38 INFO: Getting current
temperature
2023/07/29 20:54:38 INFO: Setting light
```

If running this application locally, open a browser and visit <http://localhost:3040> and you should see the “External Weather Temperature Exporter” page display with a link to the `metrics` page. Selecting that link will list the various Prometheus exporter values, one of which is the `external_weather_temperature` variable with the current weather value next to it.

Congratulations! You have created a sophisticated, scalable weather application that sets the light color to indicate the current temperature range. Now let's make it even more robust by placing it into a container that'll start whenever the Pi it's running on restarts.

Containerizing and Deploying the App

Let's deploy the application on the Raspberry Pi as a container. First, create a container image for this application by writing a **Dockerfile** using the multi-stage approach you used in [*Containing and Deploying the Exporter*](#):

[lightingweather/Dockerfile](#)

```
FROM docker.io/golang:1.22 AS builder
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build \
    -ldflags="-s -w" \
    -o lightweight \
    lightingweather.go config.go

FROM docker.io/alpine:latest
RUN mkdir /app && adduser -h /app -D lightweight
WORKDIR /app
COPY --chown=lightweight --from=builder \
/app/lightweight .
EXPOSE 3040
```

```
ENTRYPOINT ["/app/lightweather"]
```

The content of this [Dockerfile](#) is almost the same as the other [Dockerfiles](#) you've used so far in this book, but instead of using the [CMD](#) instruction to run the application, we're using the [ENTRYPOINT](#) instruction. Using the [ENTRYPOINT](#) instructions facilitates passing additional parameters to the application, which is useful in this case to specify the option `-c` to use an alternative config file, allowing you to map a file from the host when launching the container.

Save the [Dockerfile](#) and use the command `docker build` to build the container image:

```
$ docker build -t lightweather:v1 .
```

Verify the image was created correctly using `docker images`:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID
lightweather	v1	64b5c2373371
6 minutes ago	17.6MB	

To run this application and connect to the OpenWeather API, you need to provide the API key. While you can do this by

specifying it in the configuration file, there's a chance that you could commit it to a Git repository and expose the credentials. Instead, let's use an environment variable to define the key locally. First, export the environment variable on the shell:

```
$ export OWM_API_KEY="my_api_key"
```

Now, launch the container using `docker run`. Use the option `-e` to expose the environment variable `OWM_API_KEY` to the container. Use option `-v` to mount the config file `config.yml` from your host machine into the `/etc/config.yml` file in the container. Use the Prometheus network `prometheus_prom_net` to make it easier for the Prometheus container to scrape metrics from this application. Finally, append the application parameter `-c /etc/config.yml`, allowing the application to read the configuration file mounted from your local machine:

```
$ docker run -d \
  --name lightweather \
  --restart=always \
  -v $(pwd)/config.yml:/etc/config.yml \
  -p 3040:3040 \
  -e OWM_API_KEY=${OWM_API_KEY} \
  --net=prometheus_prom_net \
  lightweather:v1 \
```

```
-c /etc/config.yml
```

Ensure the container is running by using `docker ps`:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
98d505b2b798	lightweather:v1	"/app/lightweather -..."
Up 2 minutes	0.0.0.0:3040->3040/tcp	lightweather

You can also check the application logs:

```
$ docker logs -f lightweather
```

```
2023/07/30 04:14:55 INFO: Getting current
temperature
2023/07/30 04:14:55 INFO: Setting light
```

This application refreshes your light according to the external weather automatically every 30 minutes. You can force a refresh at any moment by hitting the `refresh` endpoint with a `POST` HTTP request. Test the refresh functionality using the command

`curl` like this, replacing `PI_HOST` with your Raspberry Pi hostname or IP address:

```
$ curl -s -XPOST <PI_HOST>:3040/refresh
```

You can check the refresh call succeeded by looking at the application logs using `docker logs`:

```
$ docker logs -f lightweather
...
2023/07/30 04:33:37 INFO: Received refresh request
2023/07/30 04:33:37 INFO: Getting current
temperature
2023/07/30 04:33:37 INFO: Setting light
```

Your light strip may also have changed color depending on the external temperature.

In addition to controlling the light strip, your application exposes the external temperature as a Prometheus metric. You can check it by issuing an HTTP `GET` request to the `metrics` endpoint:

```
$ curl -s <PI_HOST>:3040/metrics | grep -i weather
# HELP external_weather_temperature
# TYPE external_weather_temperature gauge
```

```
external_weather_temperature 24
```

Configure your Prometheus instance to scrape for these metrics by creating a service discovery file for it:

[lightingweather/sd_lightweather.yml](#)

```
---
- labels:
    job: lightweather
  targets:
    - 'lightweather:3040'
```

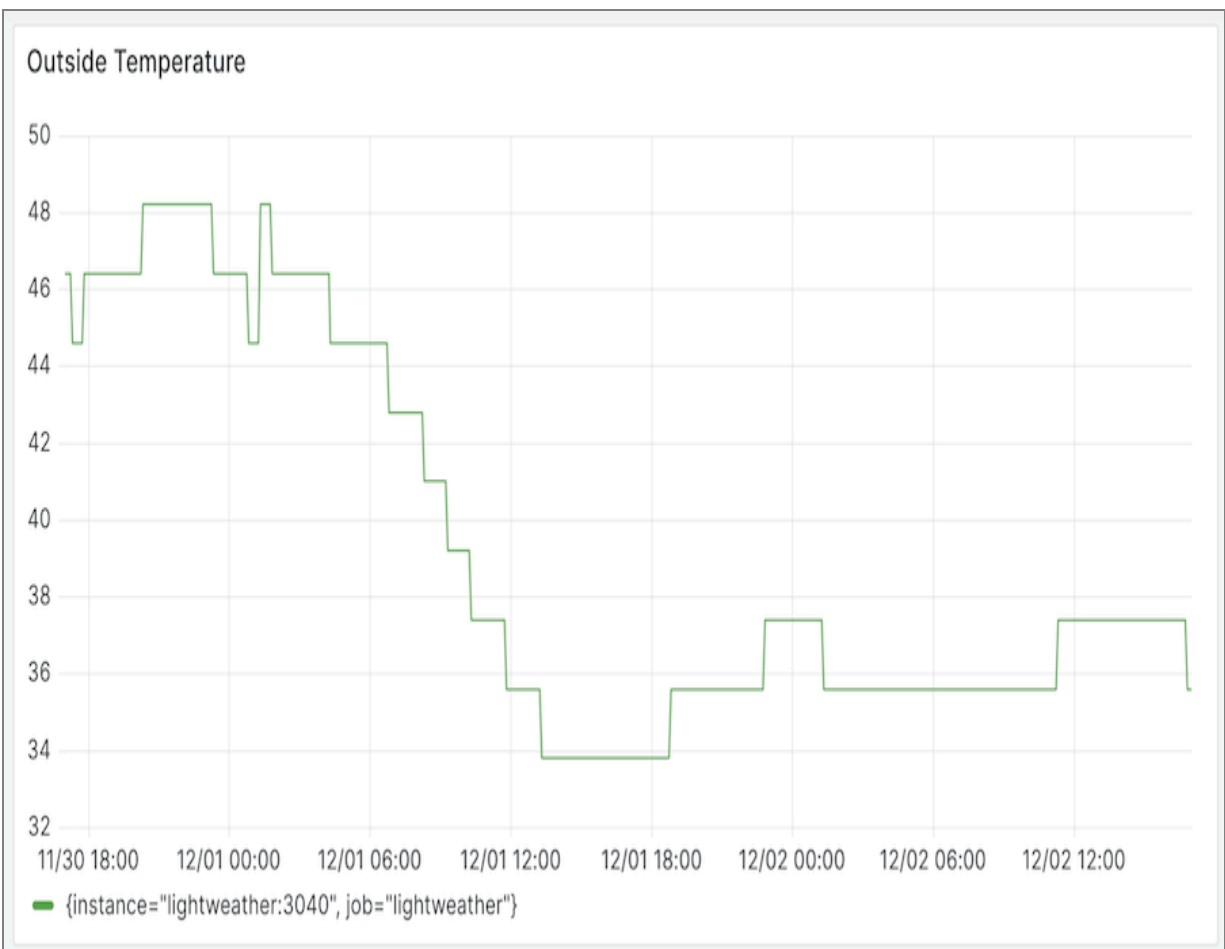
Since you're running the application on the same network as Prometheus, the Prometheus scraper can reach your application using its container name `lightweather` on port `3040`. Copy this service discovery file to the Prometheus configuration directory in the `prometheus-prometheus-1` server container:

```
$ docker cp sd_lightweather.yml prometheus-
prometheus-1:/prometheus
```

Using this file, the automatic service discovery you set up in [Monitoring and Alerting with Prometheus](#), configures a new Prometheus target for your application. Wait a few seconds,

then check the new target in the Prometheus web interface, as shown in the next image:

Because you're using the same Prometheus instance, your new metric is available in Grafana. Use the same procedure you used in [*Creating the Grafana Dashboard*](#), to create a dashboard for the external temperature, as shown in the [*picture*](#).



You can even combine the external temperature metric with the Pico W temperature and create a single dashboard that contrasts both metrics, like this:

In addition to changing the light color, you could use Grafana to alert you when the external temperature is too high or too low; for example, reminding you to drain the garden hose when the temperature reaches near freezing.

Next Steps

You can now control your Hue lighting to visually represent changing monitored values. Nice job! Imagine the other possibilities you can apply using this approach. Set up a network monitor and flash your Hue lights on and off while casting a red color anytime persistent network connections are interrupted. Couple lighting triggers with the Chapter 5, *[Checking the \(Garage\) Door](#)*, project to automatically turn on a room light whenever motion is detected. Connect to X (Twitter), Reddit, or other service APIs and pulsate a representative color anytime those services have messages waiting for you. Turn your home lighting into a sophisticated messaging center!

In the next chapter, we'll build one of the neatest projects in the book for a grand finale. See you there!

FOOTNOTES

[58]<https://openweathermap.org/>

[59]https://home.openweathermap.org/users/sign_up

[60]<https://play.google.com/store/apps/details?id=com.philips.lighting.hue2>

[61]<https://apps.apple.com/ie/app/philips-hue/id1055281310>

Chapter 7

Watching the Birds

Sometimes all you need to do to get inspiration for a project is step outside for some fresh air and take a look around. If you have trees or bushes in your yard, there are sure to be birds occasionally perching on them. Want to see them up close? Then this project is tailored for just that purpose.

In exchange for some bird seed and a sheltered place to eat them, birds will be ideal photographic subjects to point a Raspberry Pi camera at. Rather than waiting around next to a bird feeder for a bird to land and strike the perfect pose, use your Pi, a motion sensor, and some Go code to trigger the camera when the bird approaches the feeder perch to trade their image for a free meal. Automatically post these captured photos from your Pi to a Discord channel where you and your channel subscribers can see the captured images in near real time. Share the best images with friends, family, and even fellow Go Home book readers like yourself.

Project's Hardware Requirements

This project requires these components:

- *Raspberry Pi*: A Raspberry Pi 3 or higher to run the application.
- *Raspberry Pi Camera Module*:^[62] The Pi camera module is ideal since it already takes advantage of the Pi's onboard camera connector.
- *Passive InfraRed (PIR) motion sensor*:^[63] This inexpensive device has three leads: power, ground, and signal. To connect these leads to the respective GPIO pins on the Pi, you'll also need three female-to-female jumper wires.

For more details, consult [*Adding Other Hardware Components*](#).

In addition to the electronic components, you'll also need a place to house the project. A bird feeder is ideal as it offers an enclosure to contain the hardware assembly while offering a trough and a perch where birds can feed. You'll have plenty of different feeders to choose from, ranging from inexpensive clear plastic to elaborate and highly decorative designs. Choose the model that most appeals to you while still having enough room and protection for your hardware against the elements. Oh, and a few cups of birdseed will be needed as well to help attract birds to the assembly.

By the end of this chapter, you'll not only have a nice device to monitor wildlife, but you'll also have the skills required to use the Raspberry Pi camera module in a variety of home projects.

With all those items in hand, you're ready to assemble the hardware for the project.

Setting Up the Camera and InfraRed Sensor

On most Raspberry Pi models, the Pi's camera connector is located near the Ethernet and HDMI ports or, in the case of the Pi 4, between the HDMI USB-C and A/V ports, and is labeled CAMERA on the board itself. To install the camera, lift up the connector's plastic clasp and insert the end of the camera's cable ribbon into it, with the ribbon's contacts facing away from the clasp. Once the cable is seated, push down on the clasp to lock the cable in place. Verify that the camera is working by taking a test snapshot using the `rpicam` tools that are included in the latest version of Raspberry Pi OS distribution. Use the following syntax to take a snapshot.

```
$ rpicam-still -o capture.jpg
```

If you are running the Pi desktop connected to an HDMI monitor, verify that the picture was taken by opening the `capture.jpg` file on the Pi itself. Otherwise, copy the file to another computer to view it. If the image is missing or the `rpicam-still` command displays an error, check that the camera cable ribbon is properly inserted, with all the ribbon contacts correctly seated, aligned, and touching the intended contacts on the Pi.

Proceed once you have successfully captured an image with the camera.

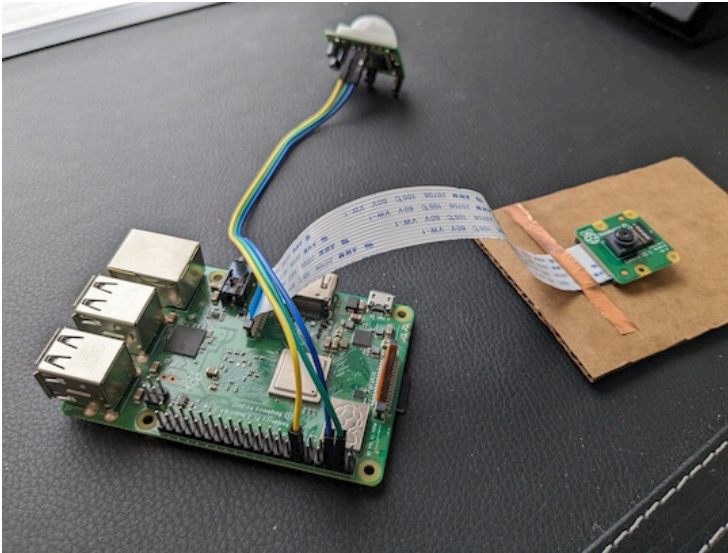
libcamera



Older versions of Raspberry PI OS use the *libcamera-apps* package to control the camera. If you have an older version of the OS, use *libcamera-still* instead of *rpicom-still* to take the snapshot. In the new OS version, *libcamera-still* is a link to *rpicom-still*, but since it has been deprecated, we recommend using the new syntax instead.

Now that the camera snapshots have been confirmed as working, it's time to attach the PIR sensor to the Pi. Using the female-to-female wires, connect the PIR's power pin to the Pi's 5V pin located on the first row, second pin from the left. Then connect the PIR's ground (GND) lead to the Pi's ground pin located on the first row, third pin from the left. Finally, connect the PIR's signal (SIG) lead to the Pi's GPIO 18 pin, located on the first row, sixth pin from the left. Refer to the following wiring diagram for details.

And the picture shows what the completed physical assembly should look like.



With the hardware fully assembled and the Pi powered up, it's time to write some Go code to detect motion whenever you wave your hand near the PIR.

Writing the Software

Before sending photos, we need to make sure we can accurately detect any movement with the PIR sensor. Fire up your code editor of choice and enter this rudimentary Go program that calls upon the status of the Raspberry Pi's GPIO pin 18 that we attached to the PIR signal (SIG) lead. Whenever a hand or a bird wing waves near the PIR sensor, we should see a signal sent to the Pi's GPIO pin that we're monitoring.

[birdwatcher/motion/main.go](https://github.com/stianeikeland/birdwatcher/blob/master/motion/main.go)

```
package main

import (
    "fmt"
    "os"
    "time"

    "github.com/stianeikeland/go-rpio/v4"
)

func main() {
    pin := rpio.Pin(18)

    if err := rpio.Open(); err != nil {
```

```
        fmt.Println(err)
        os.Exit(1)
    }

    defer rpio.Close()

    pin.Input()
    pin.PullUp()
    pin.Detect(rpio.FallEdge)

    fmt.Println("Sensing Enabled.")

    for range time.Tick(500 * time.Millisecond) {
        if pin.EdgeDetected() {
            fmt.Println("Motion detected.")
        }
    }
}
```

Compile and execute the code via the usual Go run command.

```
$ go run main.go
```

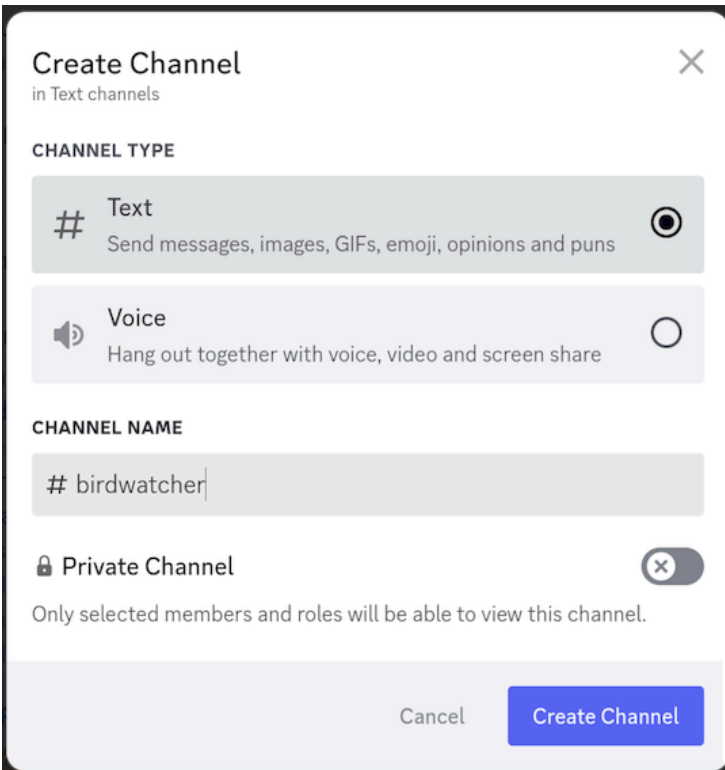
With the code running, wave your hand near the PIR. If everything is connected correctly and your code is successfully

running, “Motion Detected” should appear in the terminal’s output. Some PIR sensors can attenuate their sense arc via a small screw on the PIR’s circuit board. Refer to the documentation for your particular PIR. If you find the PIR a bit too sensitive due to the Fresnel lens spreading out the detection arc too broadly, wrap a cone of paper around it to prevent it from detecting motion, except when something moves directly in front of the PIR sensor.

Now that our hardware is correctly set up, we can enhance our Go monitoring program to call the external [rpicam-still](#) application that we used earlier to take a picture using the attached Raspberry Pi camera. We also need a convenient way to be notified that a motion detection event occurred, and copy that captured image file to a location where it can be viewed. Luckily, a very popular, free online chat service will satisfy those requirements.

Sending Motion Notifications

Similar to how you created a Discord channel for the Garage Door project in [*Sending Notifications*](#), create a new channel for this project on your Discord server by selecting the + symbol to the right of the TEXT CHANNELS label. In the pop-up, select the default #Text channel type, and name the channel something apropos of the subject matter. In this case, call the channel [*#birdwatcher*](#). If you're not ready to share your photos with the rest of the world, you can also toggle on the Private Channel option so only you can view the [*#birdwatcher*](#) channel on your Discord server. Your New Channel dialog should look similar to the one shown in the [*image*](#).

A screenshot of the 'Create Channel' dialog box in Discord. The dialog has a title bar with 'Create Channel' and a close button. Below the title, it says 'in Text channels'. The 'CHANNEL TYPE' section has two options: 'Text' (selected with a radio button) and 'Voice'. The 'Text' option has a description: 'Send messages, images, GIFs, emoji, opinions and puns'. The 'Voice' option has a description: 'Hang out together with voice, video and screen share'. The 'CHANNEL NAME' section has a text input field containing '# birdwatcher'. Below this is a 'Private Channel' toggle switch, which is currently turned on. A note below the toggle says 'Only selected members and roles will be able to view this channel.' At the bottom, there are two buttons: 'Cancel' and 'Create Channel'.

Select the [Create Channel](#) button to continue. If you selected the Private Channel option, select [Next](#) to continue to the next screen to list groups and any subscribers to your server you would like to grant access to the new [#birdwatcher](#) channel. Select those groups or individuals you want to allow to see the captures and get notified from their Discord client anytime a new capture is posted to the channel.

With the new [#birdwatcher](#) channel created and displayed in the TEXT CHANNELS listing, select the sprocket icon to the right of the [#birdwatcher](#) label to edit the channel. From this dialog box, select the Integrations menu item, followed by clicking on the [New Webhook](#) button. Give the webhook a name like [Bird Watcher](#)

Webhook and then select the **Save Changes** button. This dialog should look similar to the one shown in the next screenshot.

Integrations > Webhooks

ESC

Webhooks are a simple way to post messages from other apps and websites into Discord using internet magic.
[Learn more](#) or try [building one yourself](#).

New Webhook

POSTING TO #BIRDWATCHER

Bird Watcher Webhook

Created on 2 Dec 2023 by tuxvader

NAME

Bird Watcher Webhook

CHANNEL

#birdwatcher

Minimum Size: 128x128

Copy Webhook URL

Delete Webhook

Once you have saved your changes, select the [Copy Webhook URL](#) button to copy the newly created webhook URL. This URL will

be pasted as an environment variable in our Go program as the address to post new motion notifications and bird images.

With the active webhook URL ready for use, we now have all the pieces in place to complete the Go code for this project, so let's build the code listing. Start your program by defining the `main` package and importing the required libraries. These libraries include standard libraries such as `net/http` to call the Discord webhook, `mime/multipart` to post pictures, as well as the `go-rpio` external package to control the Pi's GPIO.

[birdwatcher/final/main.go](#)

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "mime/multipart"
    "net/http"
    "os"
    "os/exec"
    "path/filepath"
    "time"
```

```
    "github.com/stianeikeland/go-rpio/v4"  
)
```

Next, add the `main` function as the starting point for your program. This function reads the Discord webhook URL from an environment variable `DISCORD_WEBHOOK_URL`, starts the PIR sensor through the GPIO, and starts the infinite loop that triggers the image capture when the sensor detects motion.

[birdwatcher/final/main.go](https://github.com/stianeikeland/go-rpio/blob/master/main.go#L120)

```
func main() {  
    discordWebhookURL, ok := os.LookupEnv(  
        "DISCORD_WEBHOOK_URL")  
    if !ok {  
        fmt.Fprintln(  
            os.Stderr,  
            "'DISCORD_WEBHOOK_URL' env var is  
required",  
        )  
        os.Exit(1)  
    }  
  
    pin := rpio.Pin(18)
```



```
if err := rpio.Open(); err != nil {  
    fmt.Println(err)  
    os.Exit(1)  
}  
  
defer rpio.Close()  
  
pin.Input()  
pin.PullUp()  
pin.Detect(rpio.FallEdge)  
  
fmt.Println("Sensing Enabled.")  
  
perched := false  
  
for range time.Tick(500 * time.Millisecond) {  
    if pin.EdgeDetected() {  
        if perched {  
            perched = false  
            continue  
        }  
  
        perched = true  
    }  
}
```

```
        fmt.Println("Bird Perched! Taking  
snapshot")  
  
        go captureSendImage(discordWebhookURL)  
    }  
}  
}
```

Now, define the function `captureSendImage` that captures the image, then sends it to Discord. The `main` function calls this function as a `goroutine` asynchronously when the PIR sensor detects motion. This function calls three other functions and, if everything works, it deletes the picture file from the Pi at the end. If it fails to send the picture to Discord, the file stays on the Pi so you can retrieve it later.

[birdwatcher/final/main.go](#)

```
func captureSendImage(discordWebhookURL string) {  
    capture, err := captureImage()  
    if err != nil {  
        fmt.Fprintln(os.Stderr, "Error capturing  
picture:", err)  
        return  
    }  
}
```

```
    multipartReq, err :=
newMultiPartRequest(capture, discordWebhookURL)
    if err != nil {
        fmt.Fprintln(
            os.Stderr,
            "Failed to create file multipart
request:",
            err,
        )
        return
    }

    if err := sendRequest(multipartReq); err !=
nil {
        fmt.Fprintln(
            os.Stderr,
            "Capture failed to post to Discord
channel:",
            err,
        )
        return
    }
```

```
    fmt.Println("Capture posted to Discord  
channel")  
    os.Remove(capture)  
}
```

Next, add the `captureImage` function. This function creates a temporary file with a random name in the `/tmp` directory, then uses the `rpicas-still` program to capture the image and save it to the temporary file. If everything works, it returns the file name.

[birdwatcher/final/main.go](https://github.com/birdwatcher/final/main.go)

```
func captureImage() (string, error) {  
    output, err := os.CreateTemp("",  
        "capture*.jpg")  
    if err != nil {  
        return "", err  
    }  
    output.Close()  
  
    cmd := exec.Command(  
        "rpicas-still",  
        "--width", "1024",  
        "--height", "768",  
        "-o", output.Name(),  
    )
```

```

    )

    if err := cmd.Run(); err != nil {
        return "", fmt.Errorf("failed to capture
image: %s", err)
    }

    return output.Name(), nil
}

```

Then, create the function `newMultiPartRequest`. This function returns an instance of type `http.Request` which represents a request to be sent to an HTTP server, in this case, your Discord webhook. Specify the body of the request; use the `multipart` Go package to encode the captured picture file into the `multipart/form-data` format that's required to upload files to Discord.

[birdwatcher/final/main.go](#)

```

func newMultiPartRequest(path, discordWebhook
string) (*http.Request, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer f.Close()
}

```

```
body := bytes.Buffer{}
bodywriter := multipart.NewWriter(&body)

part, err := bodywriter.CreateFormFile(
    "multipart/form-data",
    filepath.Base(f.Name()),
)
if err != nil {
    return nil, err
}

io.Copy(part, f)
bodywriter.Close()

request, err :=
http.NewRequest(http.MethodPost, discordWebhook,
&body)
if err != nil {
    return nil, err
}
request.Header.Add("Content-Type",
bodywriter.FormDataContentType())
```

```
    return request, nil
}
```

Finally, define the function `sendRequest` that takes the `http.Request` instance returned by the previous function and sends it to Discord.

[birdwatcher/final/main.go](#)

```
func sendRequest(request *http.Request) error {
    client := http.Client{
        Timeout: 10 * time.Second,
    }

    response, err := client.Do(request)
    if err != nil {
        return err
    }

    defer response.Body.Close()

    if response.StatusCode != http.StatusOK {
        return fmt.Errorf(
            "invalid response from discord
channel: %s",
            response.Status,
```

```
        )  
    }  
  
    return nil  
}
```

Save the `main.go` file and run the program via the usual `go run` syntax. Set the `DISCORD_WEBHOOK_URL` environment variable to your Discord webhook URL to connect to Discord:

```
$ DISCORD_WEBHOOK_URL=<YOUR_DISCORD_WEBHOOK> go  
run main.go
```

Assuming no errors were reported during the compilation phase, your program is now running and ready for testing. Wave your hand in front of the motion sensor, and if everything works as expected, you should see a notification pop up from your Discord client. Selecting the notification should direct you to the `#birdwatcher` channel displaying the image just captured from the Raspberry Pi camera attached to your Pi.

Congratulations! You're almost ready to relocate the bird feeder from your workbench to a tree in your yard. But before we deploy it, let's place that program into a container.

Containing the Application

Now it's time to configure the program to run in an application container that automatically starts when the Pi is freshly booted. If the program encounters an error during execution, the container will automatically restart the program. First, define a [Dockerfile](#) containing the instructions to create the image, like this:

[birdwatcher/final/Dockerfile](#)

```
FROM docker.io/golang:1.22 AS builder
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build \
    -ldflags="-s -w" \
    -o birdwatcher \
    .

FROM docker.io/alpine:latest
RUN mkdir /app && adduser -h /app -D birdwatcher
WORKDIR /app
COPY --chown=birdwatcher --from=builder
/app/birdwatcher .
```

```
ENTRYPOINT ["/app/birdwatcher"]
```

Then, create the container image using **docker build**:

```
$ docker build -t birdwatcher:v1 .
```

Ensure that everything worked by listing the images available:

\$ docker images

REPOSITORY	TAG	IMAGE ID
CREATED	SIZE	
birdwatcher	v1	5977327b2c37
About a minute ago	12.3MB	

Then, run your container using `docker run`. Set the `DISCORD_WEBHOOK_URL` environment variable to your Discord webhook URL, and add the option `--restart=always` to ensure the container starts automatically when you boot your Raspberry Pi:

```
$ docker run -d \
    --name birdwatcher-v1 \
    --env DISCORD_WEBHOOK_URL=<DISCORD_URL> \
    --restart=always \
    birdwatcher:v1
```

If everything is working, you can test it by waving your hand in front of the PIR sensor and checking the picture appears on Discord. You're ready to deploy your bird feeder outside.

Configuring the Bird Feeder

Depending on the type of bird feeder you use for the project, you may need to get creative with safely securing the Pi and Pi camera inside the feeder. For example, if the feeder has a removable roof, see if you can create a makeshift shelf to slide the Pi into it. If there's not enough room to do so, you can place the Pi in a ziplock bag and pour seeds around it. This isn't ideal, since the Pi does generate heat that the bag could impede dissipating. Regardless of your configuration, you want to make sure to shield the Pi from moisture and heat, as either of these extremes will permanently damage your Pi hardware.

Position the camera to face outward toward the feeder's feeding trough. If multiple troughs exist, seal them with tape to prevent seeds from dispensing from them. This way, the only trough dispensing seeds is the one that has the camera facing the trough's respective perch. Position the PIR sensor to focus on the direction birds will likely arrive from, and power up the Pi, as shown in the next image.

The [birdwatcher](#) Go application's container will start up and be ready for testing. Wave your hand in front of the PIR and, within a couple seconds, depending on your Internet connection speeds, you should see the photo show up on your

assigned Discord channel. Once you're satisfied with the results, mount the feeder in an ideal spot where birds will likely visit. Be careful not to position the feeder where there might be leaves that rustle and branches that sway in the wind, as this motion will set off the PIR sensor, and your Discord feed will be flooded with photos. Depending on how far your exterior power plug may be from the feeder's mount, you may require an outdoor extension cord to connect the Pi's power supply.

Take extra precautions to shield the power supply from moisture or direct sunlight. Also, be aware of the Pi's operating temperatures. It's not advisable to operate the Pi's CPU in temperatures below minus 40 degrees Celsius (same as Fahrenheit) or exceeding 85 degrees Celsius (185 degrees Fahrenheit), so don't deploy the Pi-connected bird feeder in these situations where the CPU could freeze or overheat in these kinds of extreme operating environments, as shown in the next picture.

With all your hard work and preparation completed, it's time to enjoy the fruits of your labor. Watch as the Discord alerts pop up on your phone or computer showing pictures of our happy feathered friends showing their appreciation for your

generosity by posing for the camera. Depending on the species of bird, you may need to modify the position and angle of the camera to capture the ideal portrait, such as shown in the [photo](#).



It may take a couple of tweaks to set the camera position in the ideal spot, but once done, you should have some remarkable bird photos to share. Invite other bird enthusiasts to your Discord channel where they too can see the bird images you captured in real time with your Go-based Bird Feeder project!

Next Steps

Now that you know how to use the Raspberry Pi camera, post images to a dedicated Discord channel, and detect motion using a PIR sensor, you can combine these skills into a whole variety of solutions. Relocate your bird feeder or set up different types of feeders to attract other forms of wildlife, like the cute baby rabbit in this picture:

Or the selfie-posing squirrel in the following picture:

Assemble your own security system with motion detection alerts. Turn on lights whenever someone enters the room, and turn them back off after no motion is detected after a certain period of time. Create a whole unique workflow to trigger sights and sounds for a Halloween decoration, turning on and off electric motors to move costumed figures whenever a trick or treater crosses its path.

FOOTNOTES

[62]<https://www.raspberrypi.com/products/camera-module-v2/>

[63]<https://chicagodist.com/products/adjustable-infrared-pir-motion-sensor>

Chapter 8

Go Build

Nice job on completing the projects in this book! You now have a foundation to build more sophisticated home automation solutions. You can also improve your computing infrastructure to support even more robust capabilities such as virtualization, container orchestration, and even more effective monitoring and security. In addition, you can expand your knowledge of hardware via the variety of sensors and actuators that you connect to, measure, and control from your Raspberry Pi via Go code. Let's explore these ideas further.

Designing Additional Projects

Now that you know how to control appliances and monitor their changes via your own API calls, you can expand upon this framework to do even more interesting and complex solutions. Here are a few ideas to consider:

- Control your coffee machine: set up timers or motion sensors to trigger brewing times. Add a temperature sensor to your coffee pot that will alert you when your coffee has reached the perfect drinking temperature as well as when it's too cold to enjoy.
- Control other devices in your home: manage power outlets, web cameras, audio speakers, and other electrical appliances via Hue hub commands and image-capturing capabilities.
- Bring holiday decorations to life: turn a static skeleton into a dancing ghoul for Halloween, triggered anytime motion is detected. Take reaction photos and display them via a projector against a white sheet near the figure. Make winter holiday trees and lighting sparkle and pulse in time with music. Control a model train to deliver gifts and holiday announcements to visiting friends and family.

The most rewarding part of building your own solution is seeing your ideas literally come to life and easing the burden of

what was once a laborious or time-consuming task. Think about the various electronics and workflows in your own home, and how these devices and interactions can be improved with automation.

Expanding the Technologies

The Raspberry Pi hardware is remarkably flexible for a variety of computing tasks, as this book's own projects have demonstrated. However, as you continue to expand your automation solutions and begin to tax the Pi's processor beyond its capabilities, consider upgrading your server infrastructure to a more powerful and scalable solution. Mini PCs such as those from Asus, Geekom, Minisforum and others offer relatively inexpensive PCs that easily double as servers capable of hosting multiple virtual machines.

One of the most popular open source on-prem virtual machine (VM) hosting solutions is Proxmox.^[64] Proxmox will allow your PC to run and manage multiple x86-based virtual machines, allowing you to partition each server for a dedicated task. For example, one server could be used to run a Kubernetes cluster, while another could manage your messaging and alerting infrastructure. Of course, you could use an army of Pi's to do the same thing, but the ease of having multiple machines in a single, small package makes portability and equipment management much easier.

As an alternative to virtualization, you can continue to run your workloads as containers but spread and coordinate them across

multiple machines. In this book, we used Docker Compose to run our containers. Docker Compose works just fine on a single device, but what happens when you need to run multiple instances of a container across different devices? What about when you have dozens of different containers to manage? This is where a Container Management solution like Kubernetes^[65] comes to the rescue.

Kubernetes is a very flexible and scalable container orchestration platform that helps meet these objectives. Learning Kubernetes can be a bit daunting at first, as there are a lot of new ideas and technologies running under its hood. But once you have deployed and begun experimenting with your own Kubernetes cluster, managing and understanding its complexities becomes much easier.

Raspberry Pi 4s and higher are good starter hardware for running your own home-based Kubernetes cluster. Many approaches and distributions exist, but we have found the ones best optimized for the Pi are:

- K3s^[66]
- MicroK8s^[67]
- k0s^[68]

While setting up a Kubernetes cluster using any of these distributions is beyond the scope of this book, doing so is relatively easy following the simple step-by-step instructions in each project's installation documentation. And regardless of which hardware platform you use (a Pi or an x86 Proxmox-configured VM), Kubernetes can run on either of these operating systems because, like the other open source tools presented in this book, Kubernetes is written in Go!

In addition to the management platform, you can also improve the monitoring stack. Your on-prem Prometheus and Grafana instances provide a stable, scalable metrics and alert management configuration for monitoring your home automation solutions. However, there may be times when you want easier access to your dashboard.

Grafana offers a free, hosted tier of their enterprise edition that's perfect for home lab and automation enthusiasts. In addition to the same capabilities found in their on-prem open source version, Grafana Cloud provides additional alert and incident management capabilities. These include their OnCall incident response plug-in (great for being notified that a service in your home is either offline or operating outside of normal ranges), Loki for managing application and system logs, and Tempo for analyzing more granular application trace activity.

Configuring your existing on-prem Prometheus server to work with Grafana Cloud only takes a few steps. And once your metrics are in their secure cloud, you no longer have to be concerned with updates, backups, availability, and integrations since Grafana Cloud takes on those responsibilities for you.

Improving Security

Depending on how much you rely on your Internet service provider's equipment and whether or not you lease your home router from them, you may not have to worry about security as much as InfoSec professionals do at large enterprises.

Nevertheless, good security practices are always beneficial and can apply whether you have one or one thousand servers. Since your Raspberry Pi(s) will most likely be running 24x7x365, they can be vulnerable to the same flaws and exploits that other Linux distributions are dealing with. That's why it's important to keep your Pi(s) patched with the latest updates to the various applications and components that make up the operating system.

If you're running Raspberry Pi OS on your Pi(s), logging into the Pi and running `sudo apt update` followed by `sudo apt upgrade` will keep your Pi(s) up to date. Manually applying this operation can get tiresome, but open source configuration management tools like Ansible^[69] will help ease the burden of applying updates.

Ansible can do a lot more than just patching your Pi's, as it supports Windows and Mac OSs and can do everything from setting up and configuring new systems to simultaneously installing an application on multiple servers. While setting up and managing an Ansible server is outside the scope of this

book, reading the documentation^[70] is a great way to start learning and even implementing your own Ansible server and customize it for your home lab and network.

Another way to improve your network security is by using trusted certificates to encrypt communications between your computers, mobile devices, and your servers, including your Raspberry Pi's. One of the most popular, free, non-profit TLS certificate minting services used by individuals and large companies alike is Let's Encrypt.^[71]

Unlike your own self-signed certificates, Let's Encrypt's root certificate is already installed in nearly every popular web browser and OS. This makes it much easier to securely communicate with your web services on your network since you don't have to bypass scary browser security screens and programmatically set TLS communication settings to [Insecure](#) when establishing secure channels between machines.

Let's Encrypt can be slightly more work when minting your own certificates behind a firewall. It can be done manually or via automation scripts as long as you have access to editing the DNS records of a domain name. While a full discussion on how to do this is beyond the scope of this book, you can learn more about the process by searching online for step-by-step

instructions, such as those provided by blogger Akshay Jaggi^[72] for minting and deploying Let's Encrypt certificates within local area networks and home labs.

Advancing Electronics

With your home lab infrastructure built out and secured, it's time to incorporate more sophisticated autonomous interactions with objects in your home. The GPIO pins on all the Pi models allow an extensive array of sensors and actuators to be connected and controlled. Sensors range from simple motion detection to more elaborate environmental monitors such as carbon monoxide sensors and water leakage sensors. Actuators, better known as motors, can drive everything from opening doors and drawing window shades to mobilizing robotic vehicles that can autonomously provide home surveillance while you are away.

You could buy these components separately on an as-needed basis, but it may be more cost-effective and creative to buy a starter kit bundled with a mix of electronics from vendors like Amazon^[73] or Sparkfun.^[74] You'll also need to know a bit more about basic electronics, how to identify types of capacitors and resistors, and wiring and soldering. There are numerous tutorials and streaming videos on this subject, such as courses from Open University^[75] and YouTube.^[76] Once you understand the basics, you'll be able to build highly personalized solutions that specifically meet your needs and vision of the perfect home automation solution.

Having Fun

While having all this knowledge and technology at your disposal is required for a computationally sophisticated home automation environment, the most important takeaway from this book is to have fun building and maintaining your projects. Much like hobbyists who enjoy painting miniatures and displaying them, home lab tinkerers enjoy designing, building, and executing their own creations. Nothing is more satisfying and empowering than bringing an innovative, unique idea to life, and then benefiting from the improvement made to your home as a result. So have fun making great projects of your own, and share your discoveries, inventions, and learnings with others online!

FOOTNOTES

[64]<https://www.proxmox.com>

[65]<https://kubernetes.io/>

[66]<https://k3s.io/>

[67]<https://microk8s.io/>

[68]<https://k0sproject.io/>

[69]<https://www.ansible.com/>

[70]https://docs.ansible.com/ansible/latest/getting_started/index.html

[71]<https://letsencrypt.org/>

[72]<https://akshay.jaggi.co/blog/lan-certs/>

[73]https://www.amazon.com/KEYESTUDIO-Sensor-Arduino-Raspberry-Micro/dp/B016KIXSMM/ref=sr_1_3

[74]<https://www.sparkfun.com/products/21301>

[75]<https://www.open.edu/openlearn/science-maths-technology/an-introduction-electronics/content-section-0>

[76]https://www.youtube.com/results?search_query=basic+electronics+for+beginners

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to <https://pragprog.com> right now, and use the coupon code BUYANOTHER2024 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit <https://pragprog.com/become-an-author/> today to learn more and to get started.

Thank you for your continued support. We hope to hear from you again soon!

The Pragmatic Bookshelf

You May Be Interested In...

Select a cover for more information

Portable Python Projects



Discover easy ways to control your home with the powerful new Raspberry Pi hardware. Program short Python scripts that will detect changes in your home and react with the instructions you code. Use new add-on accessories to monitor a variety of measurements, from light intensity and temperature to motion detection and water leakage. Expand the base projects with your own custom additions to perfectly match your own home setup. Most projects in the book can be completed in under an hour, giving you more time to enjoy and tweak your autonomous creations. No breadboard or electronics knowledge required!

Mike Riley

(180 pages) ISBN: 9781680508598 \$45.95

Powerful Command-Line Applications in Go



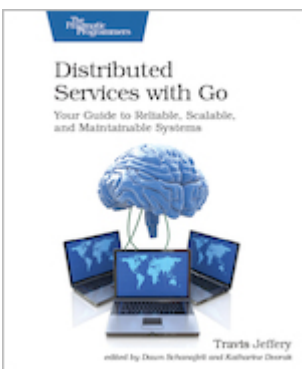
Write your own fast, reliable, and cross-platform command-line tools with the Go programming language. Go might be the fastest—and perhaps the most fun—way to automate tasks, analyze data, parse logs, talk to network services, or address other systems

requirements. Create all kinds of command-line tools that work with files, connect to services, and manage external processes, all while using tests and benchmarks to ensure your programs are fast and correct.

Ricardo Gerardi

(508 pages) ISBN: 9781680506969 \$45.95

Distributed Services with Go



This is the book for Gophers who want to learn how to build distributed systems. You know the basics of Go and are eager to put your knowledge to work. Build distributed services that are highly available, resilient, and scalable.

This book is just what you need to apply Go to real-world situations. Level up your engineering skills today.

Travis Jeffery

(258 pages) ISBN: 9781680507607 \$45.95

Effective Go Recipes



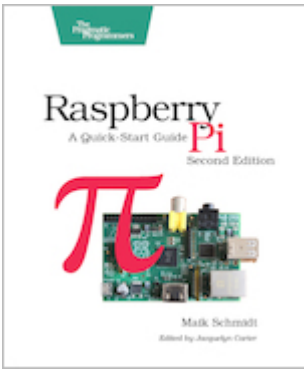
Programmers love Go because it is lightweight, easy to work with, and easy to read. Go gives you the benefits of dynamically typed languages (speed of development) while keeping the upsides of strongly typed languages (security and performance). Go is a simple language, but programming in Go is about more than just mastering syntax. There's an art to using Go effectively. Squeeze out the full use of advanced networking and multi-core power for which Go was designed. Save precious coding hours with recipes that help you manage objects, collect garbage, and safely use memory. Tackle Unicode, concurrency, and serialization with ease.

Miki Tebeka

(276 pages) ISBN: 9781680508468 \$53.95

Raspberry Pi: A Quick-Start Guide (2nd edition)

The Raspberry Pi is one of the most successful open source hardware projects ever. For less than \$40, you get a full-blown



PC, a multimedia center, and a web server—and this book gives you everything you need to get started. You'll learn the basics, progress to controlling the Pi, and then build your own electronics projects. This new edition is revised and updated with two new chapters on adding digital and analog sensors, and creating videos and a burglar alarm with the Pi camera.

Maik Schmidt

(176 pages) ISBN: 9781937785802 \$22

Arduino: A Quick-Start Guide, Second Edition



Arduino is an open-source platform that makes DIY electronics projects easier than ever. Gone are the days when you had to learn electronics theory and arcane programming languages before you could even get an LED to blink. Now, with this new edition of the bestselling Arduino: A Quick-Start Guide, readers with no electronics experience can create their first gadgets quickly. This book is up-to-date for the latest Arduino boards and for Arduino 1.x, with step-by-step

instructions for building a universal remote, a motion-sensing game controller, and many other fun, useful projects.

Maik Schmidt

(322 pages) ISBN: 9781941222249 \$34

Build Talking Apps for Alexa



Voice recognition is here at last. Alexa and other voice assistants have now become widespread and mainstream. Is your app ready for voice interaction? Learn how to develop your own voice applications for Amazon Alexa.

Start with techniques for building conversational user interfaces and dialog management. Integrate with existing applications and visual interfaces to complement voice-first applications. The future of human-computer interaction is voice, and we'll help you get ready for it.

Craig Walls

(388 pages) ISBN: 9781680507256 \$47.95

Essential 555 IC



Learn how to create functional gadgets using simple but clever circuits based on the venerable “555.” These projects will give you hands-on experience with useful, basic circuits that will aid you across other projects. These inspiring designs might even lead you to develop the next big thing. The 555 Timer Oscillator Integrated Circuit chip is one of the most popular chips in the world. Through clever projects, you will gain permanent knowledge of how to use the 555 timer will carry with you for life.

Cabe Force Satalic Atwell

(104 pages) ISBN: 9781680507836 \$19.95
