FROM SCRATCH

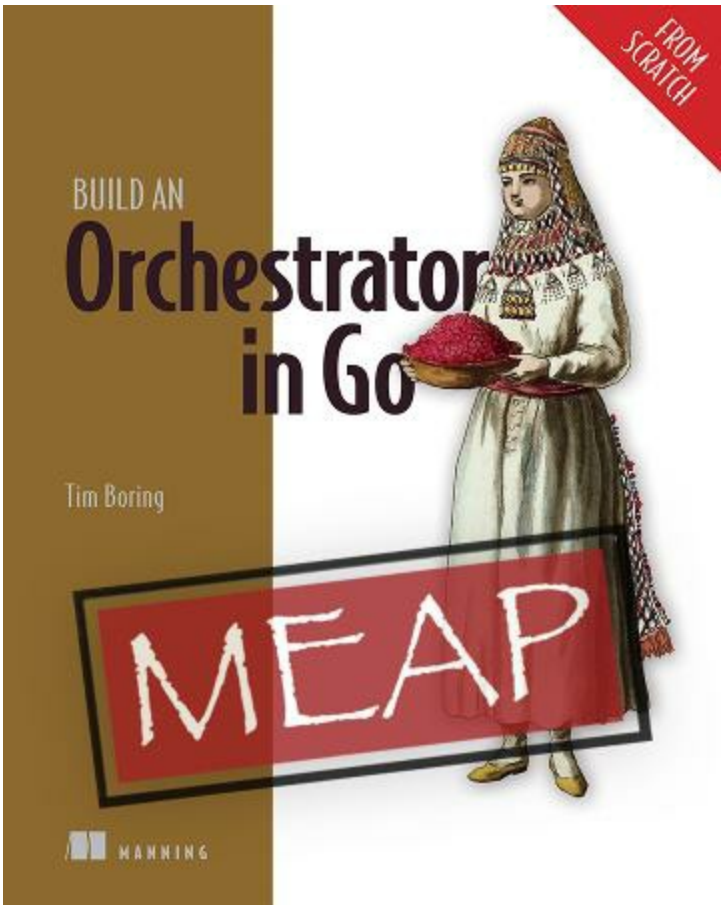
BUILD AN

Orchestrator in Go

Tim Boring

MEAP

MANNING


FROM SCRATCH
BUILD AN
Orchestrator in Go
Tim Boring
MEAP
MANNING

Build an Orchestrator in Go (From Scratch) MEAP V09

Welcome

Dear reader,

Thank you for purchasing the MEAP edition of *Build an Orchestrator in Go (From Scratch)*. I hope that you have fun learning more about orchestration systems and that it will be useful in your day-to-work.

This book is written for engineers who have experience building or running applications in Docker containers and who have at least heard of orchestration systems like Kubernetes. At a minimum you will want to be comfortable starting, stopping, and inspecting Docker containers, and you will also want to have some experiencing writing non-trivial programs. The code in this book is written in the Go programming language. If you're not familiar with Go, don't worry. The code is written with simplicity and readability in mind, so you should be able to implement the concepts in the book in any language you feel comfortable with.

You don't have to be a Kubernetes expert to get value from this book. We won't be going into the inner workings of Kubernetes, but rather discussing the concepts and components that make up an orchestration system like Kubernetes in the general sense. Thus, we won't be talking about how to containerize a web application in order to run it on Kubernetes. Instead, we'll be talking about how to write a system like Kubernetes itself.

I've been a user of orchestration systems for more than ten years. As an SRE at Google, I ran jobs on Borg. Since leaving Google, I've run jobs on both Kubernetes and Hashicorp's Nomad. Across all three of these orchestration systems, I've read documentation that dove into details about various parts of the system. While I learned some of the details, I really didn't know *how* these systems did what they did. I had guesses, of course, but that's not the same thing as knowing, at a fundamental level, how something works.

In early 2020, with the onset of the Covid-19 pandemic and finding myself working from home full-time, I suddenly had spare cycles and wanted to find

a side project. I'd been wanting to learn Go, and we were using Nomad at work to run our organizations workloads. I'd been wondering about the inner workings of orchestrators for some time, so I started by looking at Nomad's source code. I quickly realized that I wasn't interested in the inner workings of Nomad itself in so much as I wanted to understand the more general problem space. So, rather then slog through the two million plus lines of code that make up Nomad, I decided to write a clone of Nomad. For fun! While the result could in no way replace Nomad, I was able to build a functional replica in less than 5000 lines of code.

Throughout this book we'll be identifying, discussing, and implementing the core concepts of an orchestration system. By the end of the book, you will learn how to do the following:

- Identify the components that make up any orchestration system
- Start and stop containers using the Docker API
- Scheduler containers on to worker nodes
- Manage a cluster of worker nodes using a scheduler and simple API

As all of us who work in technology know, things change quickly. Understanding how orchestrators work in the general sense can help you today if you're using Kubernetes or Nomad, but it can also help you down the road if (or when?) these tools are supplanted by a newer or better version.

I hope you will have fun with this book. Along the way, if you have questions, comments, or suggestions, please post your feedback in the liveBook Discussion Forum. Your participation will help this book be the best it can be.

— Tim Boring

In this book

sophisticated scheduler 11 Implementing persistent storage for tasks

1 What is an orchestrator?

This chapter covers

- The evolution of application deployments
- Classifying the components of an orchestration system
- Introducing the mental model for the orchestrator
- Defining requirements for our orchestrator
- Identifying the scope of our work

Kubernetes. *Kubernetes*. *Kubernetes*. If you've worked in or near the tech industry in the last five years, you've at least heard the name. Perhaps you've used it in your day job. Or, perhaps you've used other systems such as Apache Mesos or Hashicorp's Nomad.

In this book, we're going to build our own Kubernetes, writing the code ourselves in order to gain a better understanding about just what Kubernetes *is*. And what Kubernetes is—like Mesos and Nomad—is an orchestrator.

When you've finished the book, you will have learned the following:

- What components form the foundation of any orchestration system
- How those components interact
- How each component maintains its own state and why
- What tradeoffs are made in designing and implementing an orchestration system

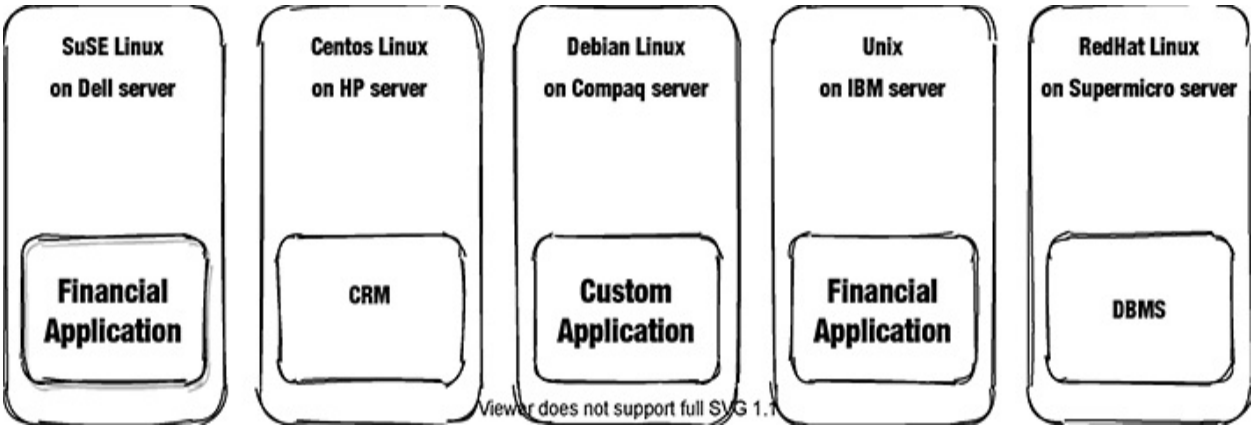
1.1 The (Not So) Good 'Ol Days

Let's take a journey back to 2002 and meet Michelle. Michelle is a system administrator for her company, and she is responsible for keeping her company's applications up and running around the clock. How does she accomplish this?

Like many other sysadmins, Michelle employs the common strategy of

deploying applications on bare metal servers. A simplistic sketch of Michelle's world can be seen in figure 1.1. Each application typically runs on its own physical hardware. To make matters more complicated, each application has its own hardware requirements, so Michelle has to buy and then manage a server fleet that is unique to each application. Moreover, each application has its own unique deployment process and tooling. The database team gets new versions and updates in the mail via compact disk, so its process involves a database administrator (DBA) copying files from the CD to a central server, then using a set of custom shell scripts to push the files to the database servers, where another set of shell scripts handles installation and updates. Michelle handles the installation and updates of the company's financial system herself. This process involves downloading the software from the Internet, at least saving her the hassle of dealing with CDs. But the financial software comes with its own set of tools for installing and managing updates. There are several other teams that are building the company's software product, and the applications that these teams build have a completely different set of tools and procedures.

Figure 1.1. This diagram represents Michelle's world in 2002. The outer box represents pysical machines and the operating systems running on them. The inner box represents the applications running on the machines and demonstrate how applications used to be more directly tied to both operating systems and machines.



If you weren't working in the industry during this time and didn't experience anything like Michelle's world, consider yourself lucky. Not only was that world chaotic and difficult to manage, it was also extremely wasteful. Virtualization came along next in the early to mid-aughts. These tools allowed sysadmins like Michelle to carve up their physical fleets so that each

physical machine hosted several smaller yet independent virtual machines (VMs). Instead of each application running on its own dedicated physical machine, it now ran on a VM. And multiple VMs could be packed onto a single physical one. While virtualization made life for folks like Michelle better, it wasn't a silver bullet.

This was the way until the mid 2010s when two new technologies appeared on the horizon. The first was Docker, which introduced *containers* to the wider world. The concept of containers was not new. It had been around since 1979 (see Ell Marquez's "The History of Container Technology"). Before Docker, containers were mostly confined to large companies, like Sun Microsystems and Google, and hosting providers looking for ways to efficiently and securely provide virtualized environments for their customers. The second new technology to appear at this time was Kubernetes, a container *orchestrator* focused on automating the deployment and management of containers.

1.2 What is a container and how is it different from a virtual machine?

As mentioned earlier, the first step in moving from Michelle's early world of physical machines and operating system was the introduction of *virtual machines*. Virtual machines, or VMs, abstracted a computers physical components (cpu, memory, disk, network, cdrom, etc.) so administrators could run multiple operating systems on a single physical machine. Each operating system running on the physical machine was distinct. Each had its own kernel, its own networking stack, and its own resources (cpu, memory, disk).

The VM world was a vast improvement in terms of cost and efficiency. The cost and efficiency gains, however, only applied to the machine and operating system layers. At the application layer, not much had changed. As you can see in figure 1.2, application were still tightly coupled to an operating system. If you wanted to run two or more instances of your application, you needed two or more VMs.

Figure 1.2. Applications running on Virtual machines.

Unlike VMs, a container does not have a kernel. It does not have its own networking stack. It does not control resources like cpu, memory, and disk. In fact, the term *container* is just a concept; it is not a concrete technical reality like virtual machine.

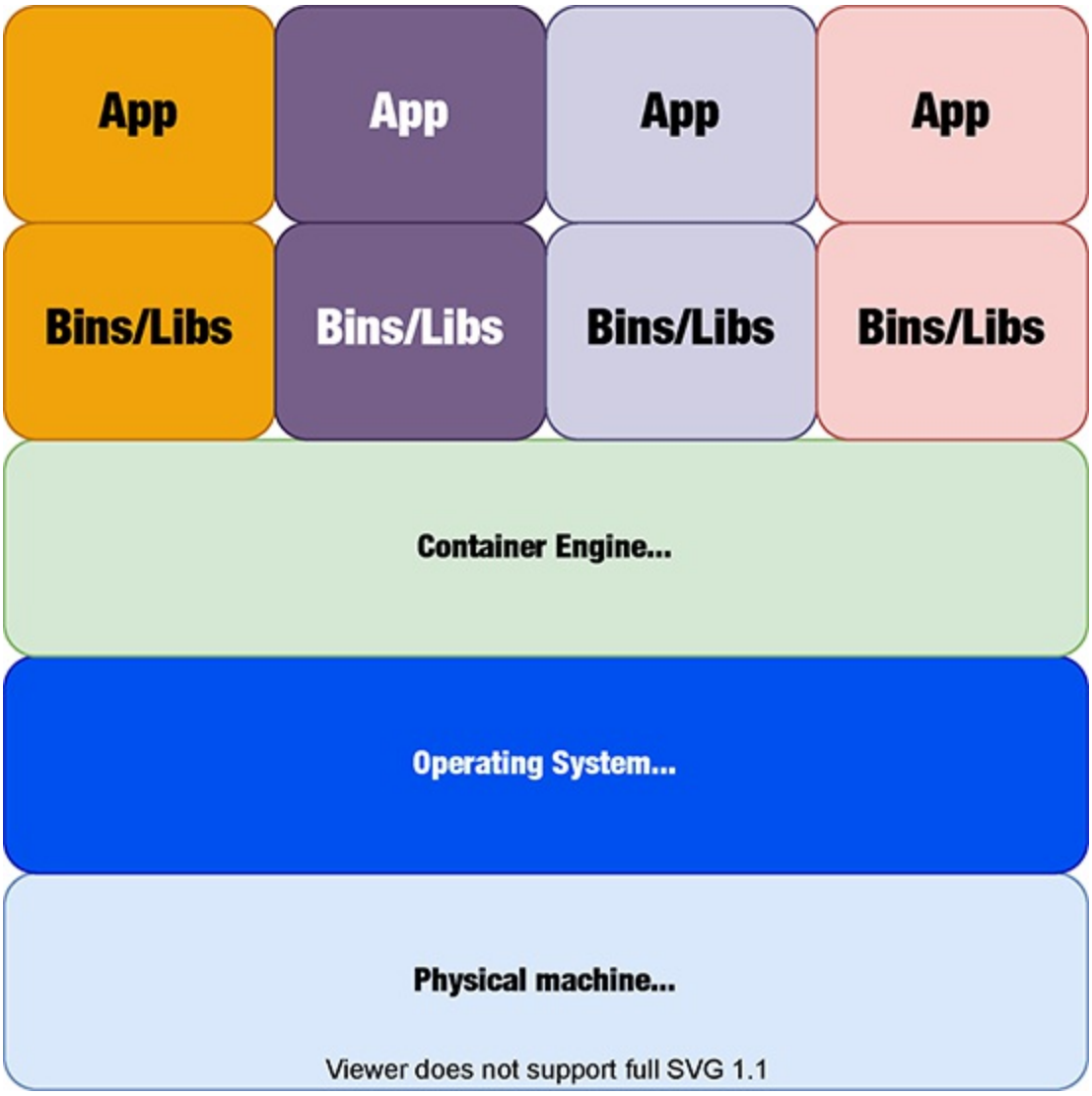
The term *container* is really just a shorthand for process and resources isolation in the Linux kernel. So, when we talk about *containers* what we really are talking about is *namespaces* and *cgroups*, both of which are features of the Linux kernel. *Namespaces* "partition resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources." (https://en.wikipedia.org/wiki/Linux_namespaces) *cgroups*, or *control*

groups, provides limits, accounting, and prioritization for a collection of processes (https://en.wikipedia.org/wiki/Cgroups)

But, let's not get too bogged down with these lower level details. You don't need to know about namespaces and cgroups in order to work through the rest of this book. If you are interested, however, I encourage you to watch Liz Rice's talk "Containers From Scratch".

With the introduction of containers, an application can be decoupled from the operating system layer, as seen in listing 1.3. With containers, if I have an app that starts up a server process that listens on port 80, I can now run multiple instances of that app on a single physical host. Or, lets say that I have six different applications, each with their own server processes listening on port 80. Again, with containers, I can run those six applications on the same host without having to give each one a different port at the application layer.

Figure 1.3. Applications running in containers.



The real benefit of containers is that it gives the application the impression it is the sole application running on the operating system, and thus has access to all of the operating system's resources.

1.3 What is an orchestrator?

The most recent step in the evolution of Michelle's world is using an *orchestrator* to deploy and manage her applications. An orchestrator is a system that provides automation for deploying, scaling, and otherwise managing containers. In many ways, an orchestrator is similar to a cpu scheduler that assigns "resources to perform tasks." (https://en.wikipedia.org/wiki/Scheduling_(computing)) The difference is that

the target object of an orchestration system is containers instead of OS-level processes. (While containers are typically the primary focus of an orchestrator, there are systems that also provide for the orchestration of other types workloads. Hashicorp's Nomad, for example, supports Java, command, and the QEMU virtual machine runner workload types in addition to Docker.)

With containers and an orchestrator, Michelle's world changes drastically. In the past, the physical hardware and operating systems she deployed and managed were mostly dictated by requirements from application vendors. Her company's financial system, for example, had to run on AIX (a proprietary Unix OS owned by IBM), which meant the physical servers had to be RISC-based (https://en.wikipedia.org/wiki/Reduced_instruction_set_computer) IBM machines. Why? Because the vendor that developed and sold the financial system certified that the system could run on AIX. If Michelle tried to run the financial system on, say, Debian Linux, the vendor would not provide support because it was not a certified OS. And this was just for one of the many applications that Michelle operated for her company.

Now, Michelle can deploy a standardized fleet of machines that all run the same OS. She no longer has to deal with multiple hardware vendors who deal in specialized servers. She no longer has to deal with administrative tools that are unique to each operating system. And, most importantly, she no longer needs the hodgepodge of deployment tools provided by application vendors. Instead, she can use the same tooling to deploy, scale, and manage all of her company's application.

Michelle's old world	Michelle's new world
Multiple hardware vendors	Single hardware vendor (or cloud provider)
Multiple operating systems	Single operating system

Runtime requirements dictated by application vendors	Application vendors build to standards (containers and orchestration)

1.4 The components of an orchestration system

So, an *orchestrator* automates deploying, scaling, managing containers. Next, let's identify the components and their requirements that make those features possible. These components can be seen in figure 1.4. They are:

- The task
- The job
- The scheduler
- The manager
- The worker
- The cluster
- The CLI

1.4.1 The Task

The *task* is the smallest unit of work in an orchestration system and typically runs in a container. You can think of it like a process that runs on a single machine. A single task could run an instance of a reverse proxy like Nginx; or it could run an instance of an application like a RESTful API server; it could be a simple program that runs in an endless loop and does something silly, like ping a website and write the result to a database.

A task should specify the following:

1. The amount of memory, CPU, and disk it needs to run effectively
2. What the orchestrator should do in case of failures, typically called a *restart policy*
3. The name of the container image used to run the task

Task definitions may specify additional details, but these are the core requirements.

1.4.2 The Job

The *job* is an aggregation of tasks. It has one or more tasks that typically form a larger logical grouping of tasks to perform a set of functions. For example, a job could be comprised of a RESTful API server and a reverse proxy.

Kubernetes and the concept of a job

If you're only familiar with Kubernetes, this definition of *job* may be confusing at first. In Kubernetesland, a *job* is a specific type of workload that has historically been referred to as a *batch job*, that is a job that starts and then runs to completion. Kubernetes actually has multiple resource types that are Kubernetes-specific implementations of the *job* concept:

- Deployment
- ReplicaSet
- StatefulSet
- DaemonSet
- Job

In the context of this book, we'll use *job* in its more common definition.

A job should specify details at a high level and will apply to all tasks it defines:

1. Each task that makes up the job
2. Which data centers the job should run in
3. How many instances of each task should run
4. The type of the job (should it be running continuously or will it run to completion and stop?)

We won't be dealing with jobs in our implementation, for the sake of simplicity. Instead, we'll work exclusively at the level of individual tasks.

1.4.3 The Scheduler

The *scheduler* decides what machine can best host the tasks defined in the

job. The decision-making process can be as simple as selecting a node from a set of machines in a round-robin fashion, or as complex as the EPVM scheduler (used as part of Google's Borg scheduler), which calculates a score based on a number of variables and then selects a node with the "best" score.

The scheduler should perform these functions:

1. Determine a set of candidate machines on which a task could run.
2. Score the candidate machines from best to worst.
3. Pick the machine with the best score.

We'll implement both the round-robin and EPVM schedulers later in the book.

1.4.4 The Manager

The *manager* is the brain of an orchestrator and the main entry point for users. In order to run jobs in the orchestration system, users submit their jobs to the manager. The manager, using the scheduler, then finds a machine where the job's tasks can run. The manager also periodically collects metrics from each of its workers, which are used in the scheduling process.

The manager should do the following:

1. Accept requests from users to start and stop tasks.
2. Schedule tasks onto worker machines.
3. Keep track of tasks, their states, and the machine on which they run.

1.4.5 The Worker

The *worker* provides the muscles of an orchestrator. It is responsible for running the tasks assigned to it by the manager. If a task fails for any reason, it must attempt to restart the task. The worker also makes metrics about its tasks and its overall machine health available for the manager to poll.

The worker is responsible for the following:

1. Running tasks as Docker containers.

2. Accepting tasks to run from a manager.
3. Providing relevant statistics to the manager for the purpose of scheduling tasks.
4. Keeping track of its tasks and their state.

1.4.6 The Cluster

The *cluster* is the logical grouping of all the above components. An orchestration cluster could be run from a single physical or virtual machine. More commonly, however, a cluster is built from multiple machines, from as few as five to as many as thousands or more.

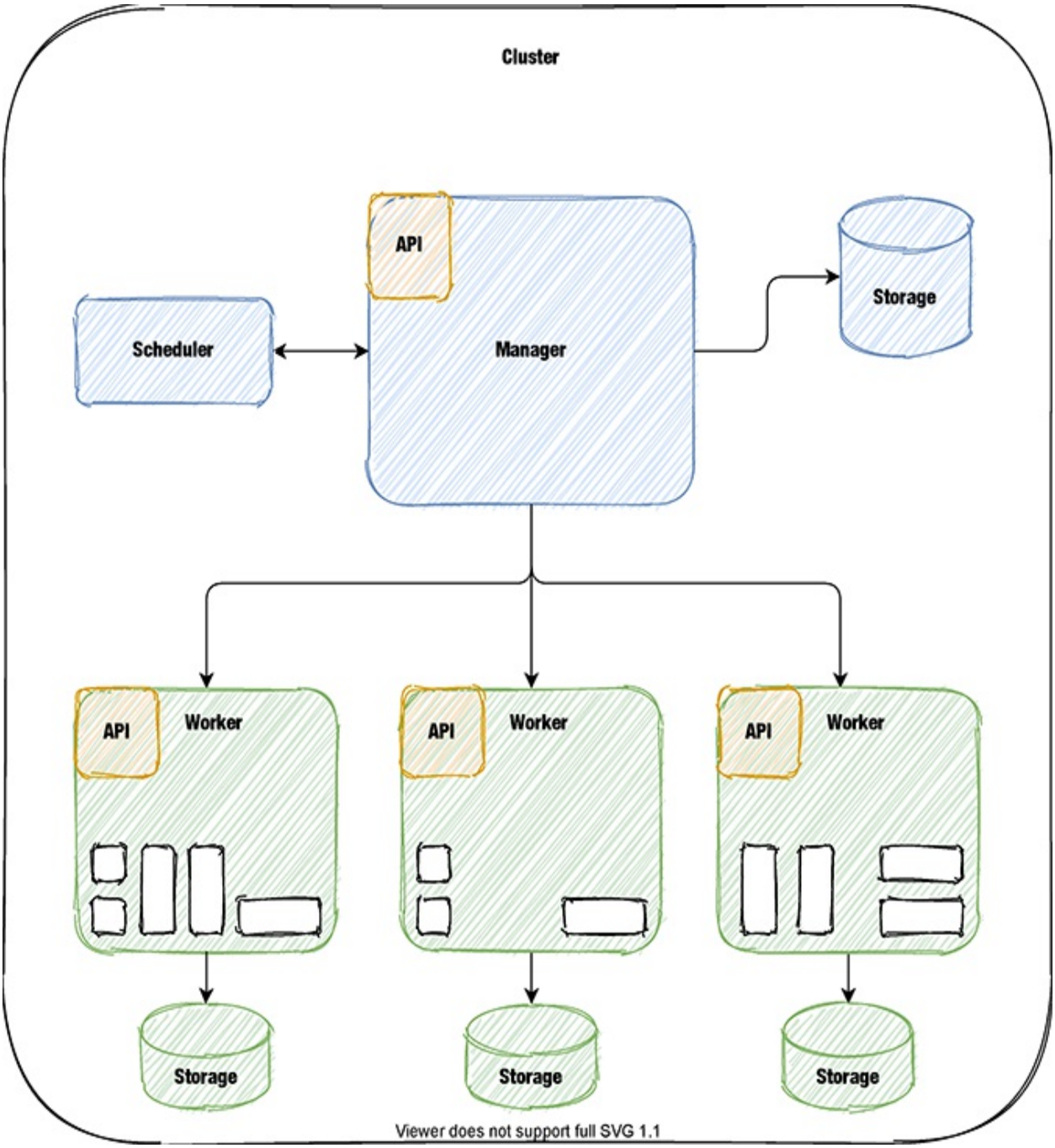
The cluster is the level at which topics like *high availability* (HA) and *scalability* come in to play. When you start using an orchestrator to run production jobs, then these topics become critical. For our purposes, we won't be discussing HA or scalability in any detail as they relate to the orchestrator we're going to build. Keep in mind, however, that the design and implementation choices we make will impact the ability to deploy our orchestrator in a way that it would meet the HA and scalability needs of a production environment.

1.4.7 CLI

Finally, our CLI, the main user interface, should allow a user to:

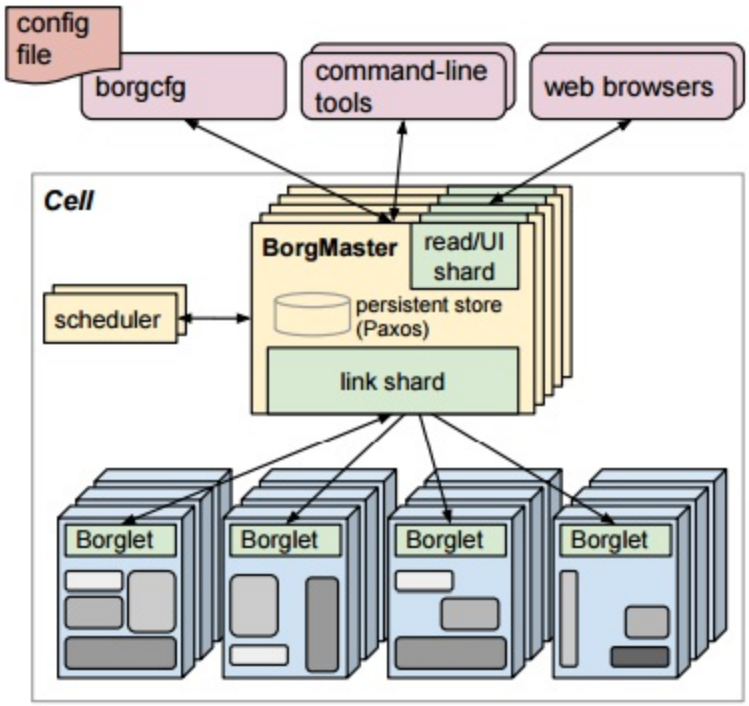
1. Start and stop tasks
2. Get the status of tasks
3. See the state of machines (i.e. the workers)
4. Start the manager
5. Start the worker

Figure 1.4. The basic components of an orchestration system. Regardless of what terms different orchestrators use, each has a scheduler, a manager, a worker, and they all operate on tasks.



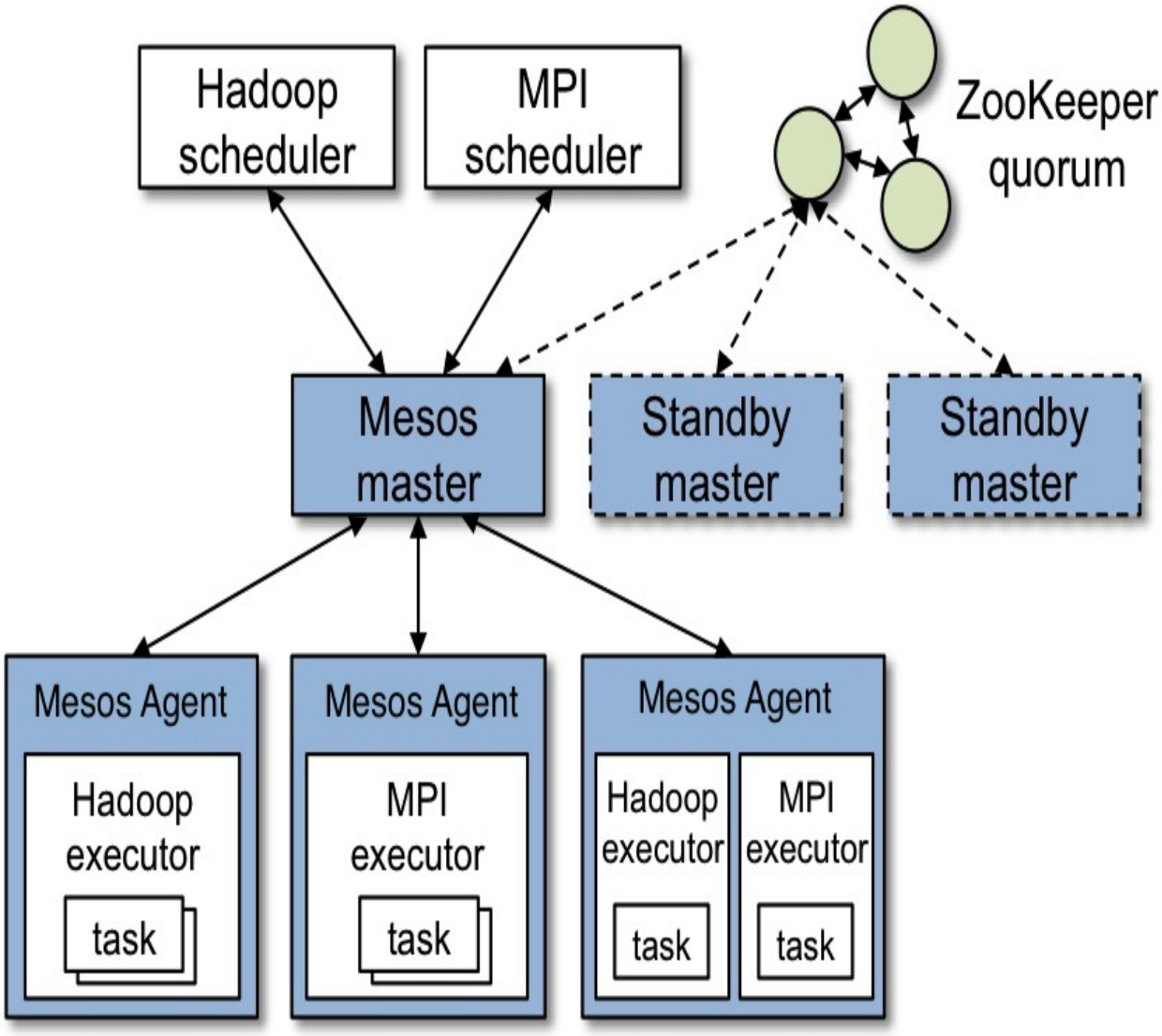
All orchestration systems share these same basic components. Google's Borg, seen in Figure 1.5, calls the manager the *BorgMaster* and the worker a *Borglet*, but otherwise uses the same terms as defined above.

Figure 1.5. Google's Borg. At the bottom are a number of Borglets, or workers, which run individual tasks in containers. In the middle is the BorgMaster, or the manager, which uses the scheduler to place tasks on workers.



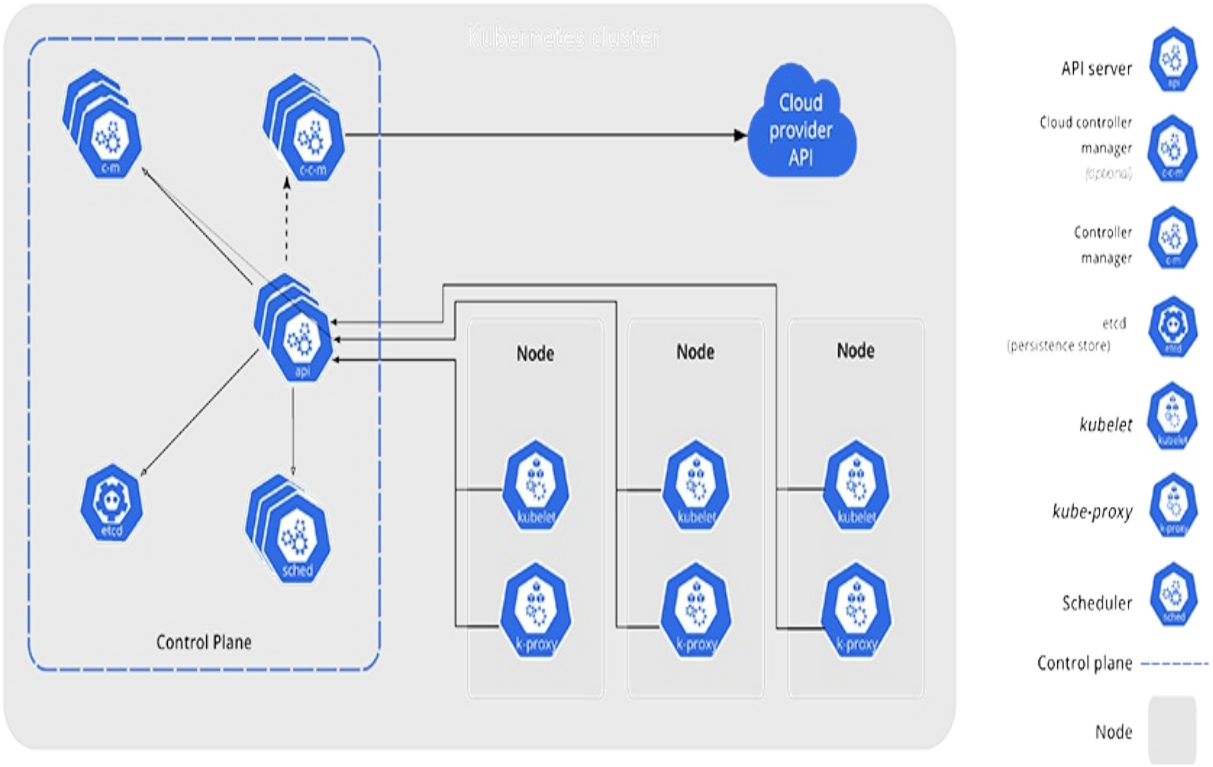
Apache Mesos, seen in figure 1.6, was presented at the Usenix HotCloud workshop in 2009 and was used by Twitter starting in 2010. Mesos calls the manager simply the *master* and the work an *agent*. It differs slightly, however, from the Borg model in how it schedules tasks. It has a concept of a *framework*, which has two components: a "*scheduler* that registers with the master to be offered resources, and an executor process that is launched on agent nodes to run the framework's tasks." (http://mesos.apache.org/documentation/latest/architecture/)

Figure 1.6. Apache Mesos



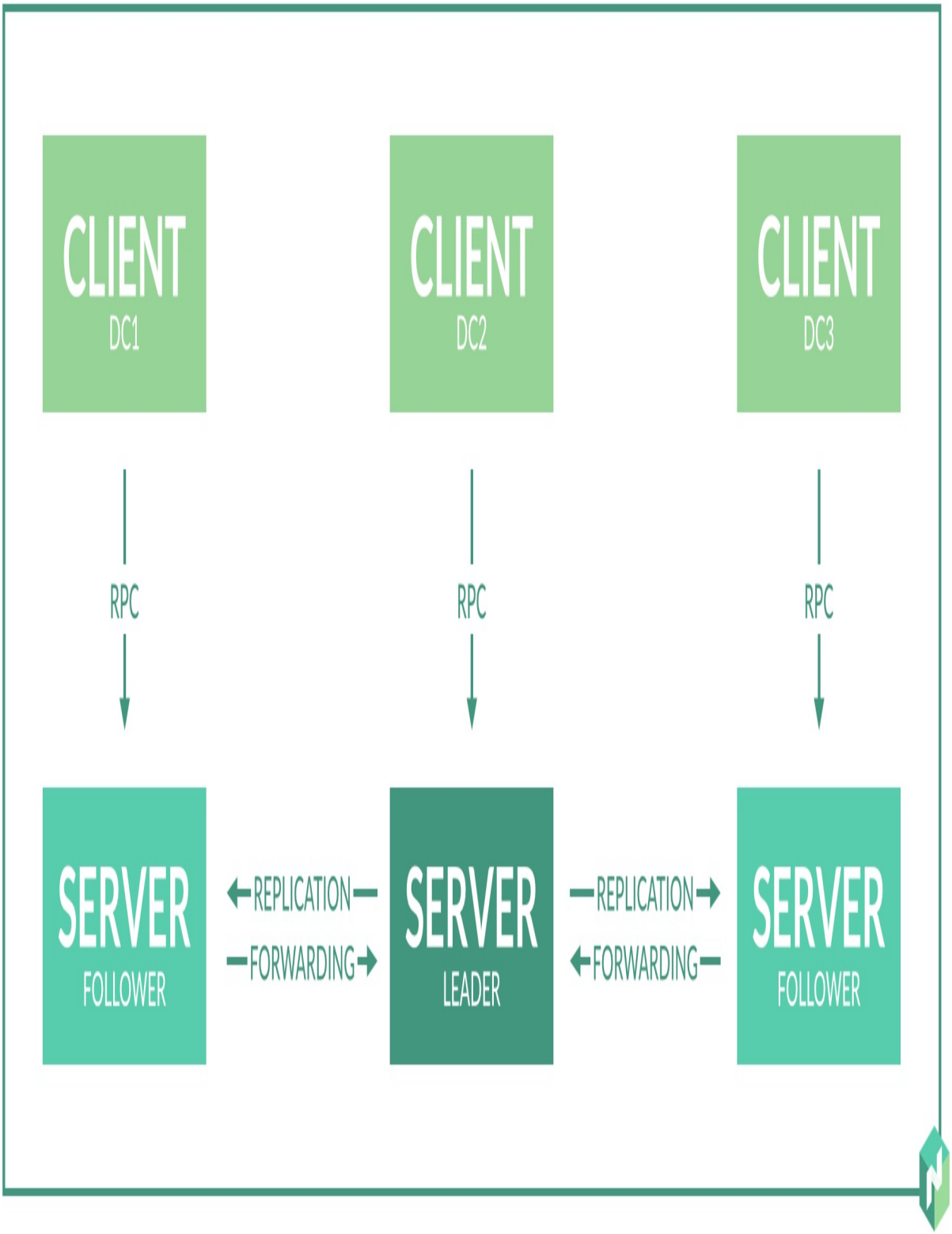
Kubernetes, which was created at Google and influenced by Borg, calls the manager the *control plane* and the worker a *kubelet*. It rolls up the concepts of job and task into *Kubernetes objects*. Finally, Kubernetes maintains the usage of the terms *scheduler* and *cluster*. These components can be seen in the Kubernetes architecture diagram in figure 1.4.

Figure 1.7. The Kubernetes architecture. The *control plane*, seen on the left, is equivalent to the manager function, or to Borg's BorgMaster.



Hashicorp's Nomad, released a year after Kubernetes, uses more basic terms. The manager is the *server*, and the worker is the *client*. While not shown in figure 1.6, Nomad uses the terms *scheduler*, *job*, *task*, and *cluster* as we've defined here.

Figure 1.8. Nomad's architecture. While it appears more sparse, it still functions similar to the other orchestrators.


CLIENT
DC1
CLIENT
DC2
CLIENT
DC3
RPC
RPC
RPC
SERVER
FOLLOWER
REPLICATION
FORWARDING
SERVER
LEADER
REPLICATION
FORWARDING
SERVER
FOLLOWER

1.5 Why implement an orchestrator from scratch?

If orchestrators such as Kubernetes, Nomad, and Mesos already exist, why write one from scratch? Couldn't we just look at the source code for them and get the same benefit?

Perhaps. Keep in mind, though, these are large software projects. Kubernetes and Nomad each contain more than 2 millions lines of source code. Mesos clocks in at just over 700,000 lines of code. While not impossible, learning a system by slogging around in codebases of this size may not be the best way.

Instead, we're going to roll up our sleeves and get our hands dirty.

To ensure we focus on the core bits of an orchestrator and don't get sidetracked, we are going to narrow the scope of our implementation. The orchestrator you write in the course of this project will be fully functional. You will be able to start and stop jobs, and interact with those jobs.

It will not, however, be production ready. After all, our purpose is not to implement a system that will replace Kubernetes, Nomad, or Mesos. Instead, our purpose is to implement a minimal system that gives us deeper insight into how production-grade systems like Kubernetes and Nomad work.

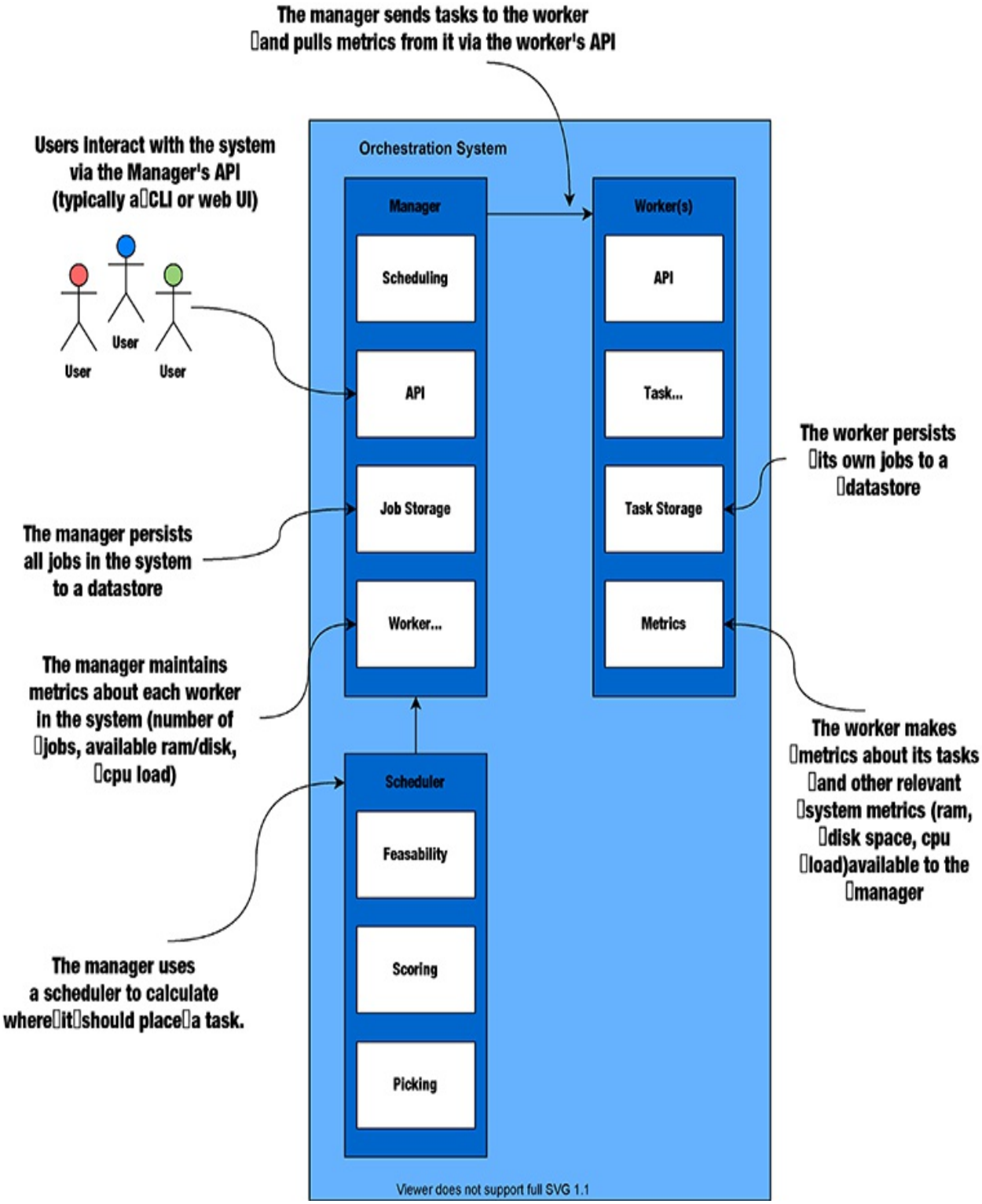
1.6 Meet Cube

We're going to call our implementation *Cube*. If you're up on your *Star Trek: Next Generation* references, you'll recall the Borg travelled in a cube-shaped spaceship.

Cube will have a much simpler design than Google's Borg, Kubernetes or Nomad. And it won't be anywhere near as resilient as the Borg's ship. It will, however, contain all the same components as those systems.

The mental model in figure 1.9 expands on the architecture outlined in figure 1.4 above. In addition to the higher level components, it dives a little deeper into the three main components: the manager, the worker, and the scheduler.

Figure 1.9. Mental model for Cube. It will have a manager, a worker, and a scheduler, and users (i.e. you) will interact with it via a command line.



Starting with the scheduler in the lower left of the diagram, we see it contains three boxes: *feasibility*, *scoring*, and *picking*. These boxes represent the scheduler's generic phases, and they are arranged in the order in which the scheduler moves through the process of scheduling tasks onto workers.

- *Feasability* This phase assesses whether it's even possible to schedule a task onto a worker. There will be cases where a task cannot be scheduled onto any worker; there will also be cases where a task can be scheduled but only onto a subset of workers. We can think of this phase similar to choosing which car to buy. My budget is $10,000, but depending on which car lot I go to all the cars on the lot could cost more than $10,000 or there may only be subset of cars that fit into my price range.
- *Scoring* This phase takes the workers identified by the feasability phase and gives each one a score. This stage is the most important and can be accomplished any number of ways. For example, to continue our car purchase analogy, I might give a score for each of three cars that fit within my budget based on variables like fuel efficiency, color, and safety rating.
- *Picking* The phase is the simplest. From the list of scores, the scheduler picks the best one. This will be either the highest or lowest score.

Moving up the diagram we come to the manager. The first box inside the manager component shows that the manager uses the scheduler we described previously. Next, there is the *API* box. The API is the primary mechanism for interacting with Cube. Users submit jobs and request jobs be stopped via the API. A user can also query the API to get information about job and worker status. Next, there is the *Job Storage* box. The manager must keep track of all the jobs in the system in order to make good scheduling decisions, as well as to provide answers to user queries about job and worker statuses. Finally, the manager also keeps track of worker metrics, such as the number of jobs a worker is currently running, how much memory it has available, how much load is the CPU under, and how much disk space is free. This data, like the data in the job storage layer, is used for scheduling.

The final component in our diagram is the worker. Like the manager, it too has an API, though it serves a different purpose. The primary user of this API

is the manager. The API provides the means for the manager to send tasks to the worker, to tell the worker to stop tasks, and to retrieve metrics about the worker's state. Next, the worker has a *task runtime*, which in our case will be Docker. Like the manager, the worker also keeps track of the work it is responsible for, which is done in the *Task Storage* layer. Finally, the worker provides metrics about its own state, which it makes available via its API.

1.7 What tools will we use?

In order to focus on our main goal, we're going to limit the number of tools and libraries we use. Here's the list of tools and libraries we're going to use:

- Go (v1.16)
- chi (v5.0.3)
- Docker SDK (v20.10.7+incompatible)
- BoltDB (v1.3.1)
- goprocinfo
- Linux

As the title of this book says, we're going to write our code in the Go programming language. Both Kubernetes and Nomad are written in Go, so it is obviously a reasonable choice for large scale systems. Go is also relatively lightweight, making it easy to learn quickly. If you haven't used Go before but have written non-trivial software in languages such as C/C++, Java, Rust, Python, or Ruby, then you should be fine. Chapter 2 will provide an introduction to the language constructs that we'll use throughout the rest of the book. If you want more in-depth material about the Go language, either *The Go Programming Language* (www.gopl.io/) or *Get Programming with Go* (www.manning.com/books/get-programming-with-go) are good resources. That said, all the code presented will compile and run, so simply following along should also work.

There is no particularly requirement for an IDE to write the code. Any text editor will do. Use whatever you're most comfortable with and makes you happy.

We'll focus our system on supporting Docker containers. This is a design

choice. We could broaden our scope so our orchestrator could run a variety of jobs: containers, standalone executables, or Java JARs. Remember, however, our goal is not to build something that will rival existing orchestrators. This is a learning exercise. By narrowing our scope to focus solely on Docker containers will help us reach our learning goals more easily. This said, we will be using Docker's Go SDK (https://pkg.go.dev/github.com/docker/docker/client).

Our manager and worker are going to need a datastore. For this purpose, we're going to use BoltDB (https://github.com/boltdb/bolt), an embedded key/value store. There are two main benefits to using Bolt. First, by being embedded within our code, we don't have to run a database server. This feature means neither our manager nor our workers will need to talk across a network to read or write its data. Second, using a key/value store provides fast, simple access to our data.

The manager and worker will each provide an API to expose their functionality. The manager's API will be primarily user-facing, allowing users of the system to start and stop jobs, review job status, and get an overview of the nodes in the cluster. The worker's API is internal-facing and will provide the mechanism by which the manager sends jobs to workers and retrieves metrics from them. In many other languages, we might use a web framework to implement such an API. For example, if we were using Java we might use Spring. Or, if we were using Python we might choose Django. While there are such frameworks available for Go, they aren't always necessary. In our case, we don't need a full web framework like Spring or Django. Instead, we're going to use a lightweight router called chi (https://github.com/go-chi/chi). We'll write handlers in plain Go, and assign those handlers to routes.

To simplify the collection of worker metrics, we're going to use the *goprocinfo* library (https://github.com/c9s/goprocinfo). This library will abstract away some details related to getting metrics from the proc filesystem.

Finally, while you can write the code in this book on any operating system, it will need to be compiled and run on Linux. Any recent distribution should be sufficient.

For everything else, we'll rely on Go and its standard tools that are installed by default with every version of Go. Since we'll be using Go modules, you should use Go 1.14 or later. I've developed the code in this book using 1.16.

1.8 A word about hardware

You won't need a bunch of hardware to complete this book. You can do everything on a single machine, whether that's a laptop or a desktop or even a Raspberry Pi. The only requirements are that the machine is running Linux, and it has enough memory and disk to hold the source code and compile it.

If you are going to do everything on a single machine, there are a couple more things to consider. You can run a single instance of the worker. This means when you submit a job to the manager, it will assign that job to the single worker. For that matter, any job will be assigned to that worker. For a better experience, and one that better exercises the scheduler and showcases the work you're going to do, you can run multiple instances of the worker. One way to do this is to simply open multiple terminals, and run an instance of the worker in each. Alternatively, you can use something like Tmux (https://github.com/tmux/tmux), seen in figure 1.7, which achieves a similar outcome but allows you to detach from the terminal and leave everything running.

Figure 1.10. A tmux session showing 3 Raspberry Pis running the Cube worker.

```
tjb@pi-1:~ $ ./cube worker
Starting worker.
2021/06/13 20:18:35 Create new worker: Worker task queue - 0
2021/06/13 20:18:35 No tasks to process currently.
2021/06/13 20:18:35 Sleeping for 10 seconds.
2021/06/13 20:18:35 Updating task count
2021/06/13 20:18:35 Collecting stats
2021/06/13 20:18:45 No tasks to process currently.
2021/06/13 20:18:45 Sleeping for 10 seconds.
2021/06/13 20:18:50 Updating task count
2021/06/13 20:18:50 Collecting stats
2021/06/13 20:18:55 No tasks to process currently.
2021/06/13 20:18:55 Sleeping for 10 seconds.
2021/06/13 20:19:05 Updating task count
2021/06/13 20:19:05 No tasks to process currently.
2021/06/13 20:19:05 Sleeping for 10 seconds.
2021/06/13 20:19:05 Collecting stats
2021/06/13 20:19:15 No tasks to process currently.
2021/06/13 20:19:15 Sleeping for 10 seconds.
2021/06/13 20:19:20 Updating task count
2021/06/13 20:19:20 Collecting stats
2021/06/13 20:19:25 No tasks to process currently.
2021/06/13 20:19:25 Sleeping for 10 seconds.
```

```
tjb@pi-2:~ $ ./cube worker
Starting worker.
2021/06/13 20:18:35 Create new worker: Worker task queue - 0
2021/06/13 20:18:35 Collecting stats
2021/06/13 20:18:35 No tasks to process currently.
2021/06/13 20:18:35 Sleeping for 10 seconds.
2021/06/13 20:18:35 Updating task count
2021/06/13 20:18:45 No tasks to process currently.
2021/06/13 20:18:45 Sleeping for 10 seconds.
2021/06/13 20:18:50 Updating task count
2021/06/13 20:18:50 Collecting stats
2021/06/13 20:18:55 No tasks to process currently.
2021/06/13 20:18:55 Sleeping for 10 seconds.
2021/06/13 20:19:05 No tasks to process currently.
2021/06/13 20:19:05 Sleeping for 10 seconds.
2021/06/13 20:19:05 Updating task count
2021/06/13 20:19:05 Collecting stats
2021/06/13 20:19:15 No tasks to process currently.
2021/06/13 20:19:15 Sleeping for 10 seconds.
2021/06/13 20:19:20 Updating task count
2021/06/13 20:19:20 Collecting stats
2021/06/13 20:19:25 No tasks to process currently.
2021/06/13 20:19:25 Sleeping for 10 seconds.
```

```
tjb@pi-3:~ $ ./cube worker
Starting worker.
2021/06/13 20:18:35 Create new worker: Worker task queue - 0
2021/06/13 20:18:35 Updating task count
2021/06/13 20:18:35 No tasks to process currently.
2021/06/13 20:18:35 Sleeping for 10 seconds.
2021/06/13 20:18:35 Collecting stats
2021/06/13 20:18:45 No tasks to process currently.
2021/06/13 20:18:45 Sleeping for 10 seconds.
2021/06/13 20:18:50 Updating task count
2021/06/13 20:18:50 Collecting stats
2021/06/13 20:18:55 No tasks to process currently.
2021/06/13 20:18:55 Sleeping for 10 seconds.
2021/06/13 20:19:05 Updating task count
2021/06/13 20:19:05 No tasks to process currently.
2021/06/13 20:19:05 Sleeping for 10 seconds.
2021/06/13 20:19:05 Collecting stats
2021/06/13 20:19:15 No tasks to process currently.
2021/06/13 20:19:15 Sleeping for 10 seconds.
2021/06/13 20:19:20 Updating task count
2021/06/13 20:19:20 Collecting stats
2021/06/13 20:19:25 No tasks to process currently.
2021/06/13 20:19:25 Sleeping for 10 seconds.
```

```
[cube]  0:ssh*                                                    "tjb@borg:~" 15:19 13-Jun-21
```

If you have extra hardware lying around—e.g. an old laptop or desktop, or a couple of Raspberry Pis—you can use those as your worker nodes. Again, the only requirement is they are running Linux. For example, in developing the code in preparation for writing this book, I used eight Raspberry Pis as workers. I used my laptop as the manager.

1.9 What we won't be implementing or discussing

So, to reiterate, our purpose here is not to build something that can be used to replace a production-grade system like Kubernetes. Engineering is about weighing trade-offs against your requirements. This is a learning exercise to gain a better understanding of how orchestrators, in general, work. To that end, we won't be dealing with or discussing any of the following that might accompany discussions of production-grade systems:

- Distributed computing
- Service discovery
- High availability
- Load balancing
- Security

1.9.1 Distributed computing

Distributed computing is an architectural style where a system's components run on different computers, communicate across a network, and have to coordinate actions and state. The main benefits of this style are scalability and resiliency to failure. An orchestrator is a distributed system. It allows engineers to scale systems beyond the resources of a single computer, thus enabling those systems to handle larger and larger workloads. An orchestrator also provides resiliency to failure by making it relatively easy for engineers to run multiple instances of their services and for those instances to be managed in an automated way.

That said, we won't be going into the theory of distributed computing. If you're interested in that topic specifically, there are many resources that cover the subject in detail.

Resources on distributed computing:

- *Designing Data-Intensive Applications* (https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/)
- *Designing Distributed Systems* (https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/)

1.9.2 Service discovery

Service discovery provides a mechanism for users, either human or other machines, to discover service locations. Like all orchestration systems, Cube will allow us to run one or more instances of a task. When we ask Cube to run a task for us, we cannot know in advance where Cube will place the task, i.e. on which worker the task will run. If we have a cluster with three worker nodes, a task can potentially be scheduled onto any one of those three nodes.

To help find tasks once they are scheduled and running, we can use a service discovery system, for example Consul (www.consul.io) to answer queries about how to reach a service. While service discovery is indispensable in larger orchestration systems, it won't be necessary for our purposes.

Resources on service discovery:

- *Service Discovery in a Microservices Architecture* (https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/)

1.9.3 High availability

The term *availability* refers to the amount of time that a system is available for usage by its intended user base. Often you'll hear the term *High Availability* (HA) used, which refers to strategies to maximize the amount of time a system is available for its users. Several examples of HA strategies are:

- elimination of single points of failure via redundancy
- automated detection of and recovery from failures
- isolation of failures to prevent total system outages

An orchestration system, by design, is a tool that enables engineers to implement these strategies. By running more than one instance of, say, a mission critical web API, I can ensure the API won't become completely unavailable for my users if a single instance of it goes down for some reason. By running more than one instance of my web API on an orchestrator, I ensure that if one of the instances does fail for some reason, the orchestrator will detect it and attempt to recover from the failure. If any one instance of my web API fails, that failure will not affect the other instances (with some exceptions, continue reading below).

At the same time, it is common to use these strategies for the deployment of the orchestration system itself. Production orchestration systems typically use multiple worker nodes. For example, worker nodes in a Borg cluster number in the tens of thousands. By running multiple worker nodes, the system permits users like me to run multiple instances of my mission critical web API across a number of different machines. If one of those machines running my web API experiences a catastrophic failure (maybe a mouse took up residence in the machine's rack and accidentally unseated the machine's power cord), my application can still serve its users.

For our purposes in this book, we will implement our orchestrator so multiple instances of the worker can be easily run in a manner similar to Google's Borg. For the manager, however, we will only run a single instance. So, while our workers can be run in an HA way, our manager cannot. Why?

The manager and worker components of our orchestration system—of any orchestration system—have different scopes. The worker's scope is narrow, concerned only with the tasks that it is responsible for running. If worker #2 fails for some reason, worker #1 doesn't care. Not only does it not care, it doesn't even know worker #2 exists.

The manager's scope, however, encompasses the entire orchestration cluster. It maintains state for the cluster: how many worker nodes there are, the state of each worker (cpu/memory/disk capacity as well as how much of that

capacity is already being used), and the state of each task submitted by users. In order to run multiple instances of the manager, there are many more questions to ask:

- Among the manager instances, will there be a *leader* that will handle all of the management responsibilities; or can any manager instance handle those responsibilities?
- How are state updates replicated to each instance of the manager?
- If state data gets out of sync, how do the managers decide which data to use?

These questions ultimately lead to the topic of *consensus*, which is a fundamental problem in distributed systems. While this topic is interesting, it isn't critical to our learning about and understanding how an orchestration system works. If our manager goes down, it won't effect our workers. They will continue to run the tasks already assigned to them. It does mean our cluster won't be able to accept new tasks, but our purposes, we're going to decide that this is acceptable for the exercise at hand.

Resources on high availability:

- https://www.freecodecamp.org/news/high-availability-concepts-and-theory/
- https://livebook.manning.com/book/learn-amazon-web-services-in-a-month-of-lunches/chapter-14

Resources on consensus:

- https://people.cs.rutgers.edu/~pxk/417/notes/content/consensus.html
- *Paxos Made Simple* (https://lamport.azurewebsites.net/pubs/paxos-simple.pdf)
- *The Raft Consensus Algorithm* (https://raft.github.io/)

1.9.4 Load balancing

Load balancing is a strategy for building highly available, reliable, and responsive applications. Common load balancers (LBs) include Nginx, HAProxy, and AWS's assortment of load balancers (classic ELBs, network

LBs, and the newer application LBs). While they are used in conjunction with orchestrators, they can become complex quickly because they are typically employed in multiple ways.

For example, it's common to have a public-facing LB that serves as an entrypoint to a system. This LB might know about each node in an orchestration system, and it will pick one of the nodes to which it forwards the request. The node receiving this request is itself running a LB that is integrated with a service discovery system and can thus forward the request to a node in the cluster running a task that can handle the request.

Load balancing as a topic is also complex. It can be as simple as using a round-robin algorithm, in which the LB simply maintains a list of nodes in the cluster and a pointer to the last node selected. When a request comes in, the LB simply selects the next node in the list. Or, it can be as complex as choosing a node that is best able to meet some criteria, such as the resources available or the lowest number of connections.

While load balancing is an important tool in building highly available production systems, it is not a fundamental component of an orchestration system.

Resources on load balancing:

- http://cbonte.github.io/haproxy-dconv/2.5/intro.html#2
- https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/

1.9.5 Security

Security is like an onion. It has many layers, many more than we can reasonable cover in this book. If we were going to run our orchestrator in production, we would need to answer questions like:

- How do we secure the manager so only authenticated users can submit tasks or perform other management operations?
- Should we use authorization in order to segment users and the operations they can perform?

- How do we secure the workers so they only accept requests from a manager?
- Should network traffic between the manager and worker be encrypted?
- How should the system log events in order to audit who did what and when?

Resources on security:

- *API Security in Action* (https://www.manning.com/books/api-security-in-action)
- *Security by Design* (https://www.manning.com/books/secure-by-design)
- *Web Application Security* (https://www.oreilly.com/library/view/web-application-security/9781492053101/)

In the next chapter we're to start coding by translating our mental model into skeleton code.

1.10 Summary

- Orchestrators abstract machines and operating systems away from developers, thus leaving them to focus on their application.
- An orchestrator is a system comprised of a manager, worker, and scheduler. The primary objects of an orchestration system are tasks and jobs.
- Orchestrators are operated as a cluster of machines, with machines filling the roles of manager and worker.
- In orchestration systems, applications typically run in containers
- Orchestrators allow for a level of standardization and automation that was difficult to achieve previously.

2 From mental model to skeleton code

This chapter covers

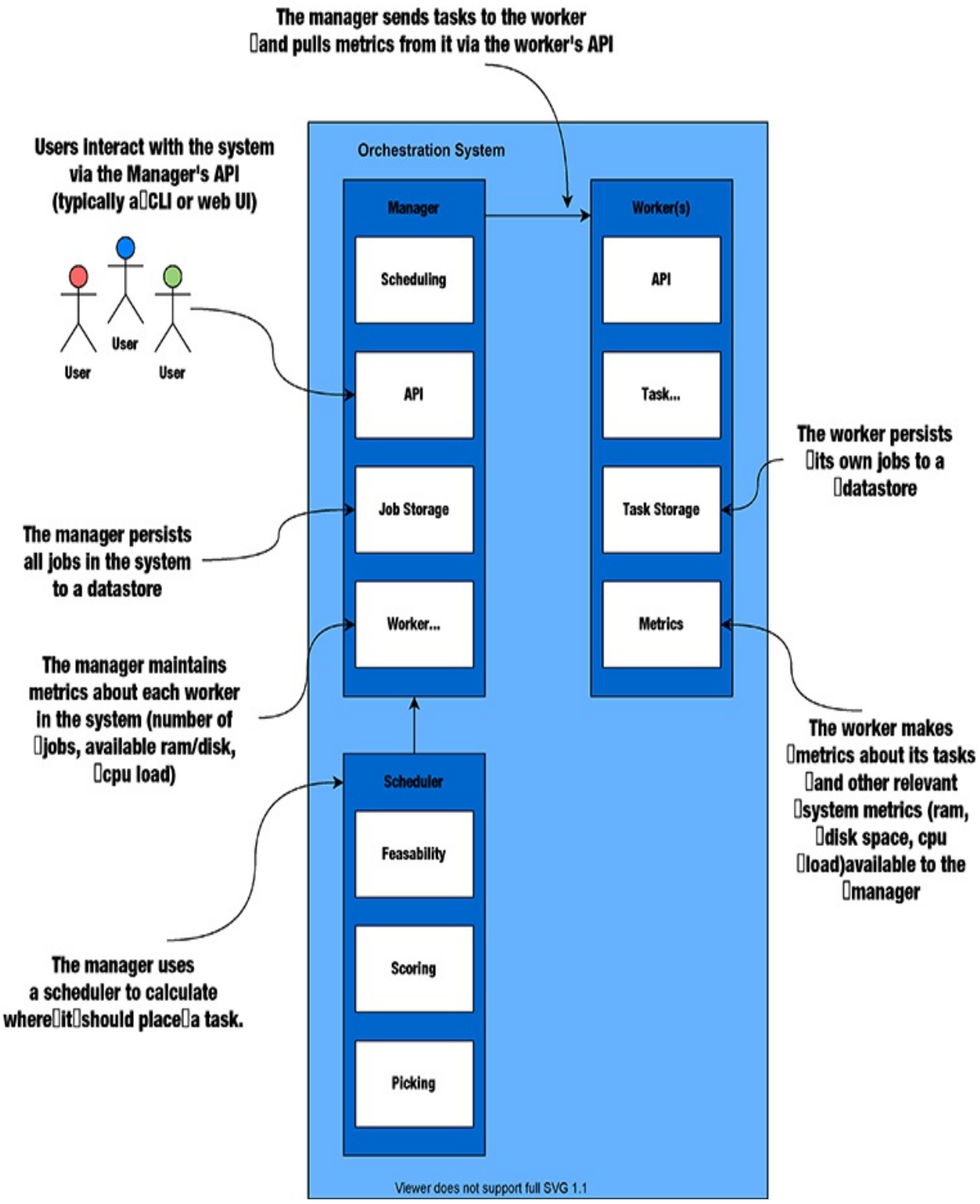
- Creating skeletons for the task, worker, manager, and scheduler components
- Identifying the states of a task
- Using an interface to support different types of schedulers
- Writing a test program to verify the code will compile and run

Once I have a mental model for a project, I like to translate that model into skeleton code. I do this quite frequently in my day job. It's similar to how carpenters frame a house: they define the exterior walls and interior rooms with 2x4s, add trusses to give the roof a shape. This frame isn't the finished product, but it marks the boundaries of the structure, allowing others to come along and add details at a later point in the construction.

In the same way, *skeleton code* provides the general shape and contours of the system I want to build. The final product may not conform exactly to this skeleton code. Bits and pieces may change, new pieces may be added or removed, and that's okay. This typically allows me to start thinking about the implementation in a concrete way without getting too deep into the weeds just yet.

If we look again at our mental model (Figure 2.1), where should we start? You can see immediately that three most obvious components are the *manager*, *worker*, and *scheduler*. The foundation of each of these components, however, is the *task*, so let's start with it.

Figure 2.1. The Cube mental model shows the Manager, Worker, and Scheduler as the major components of the system.



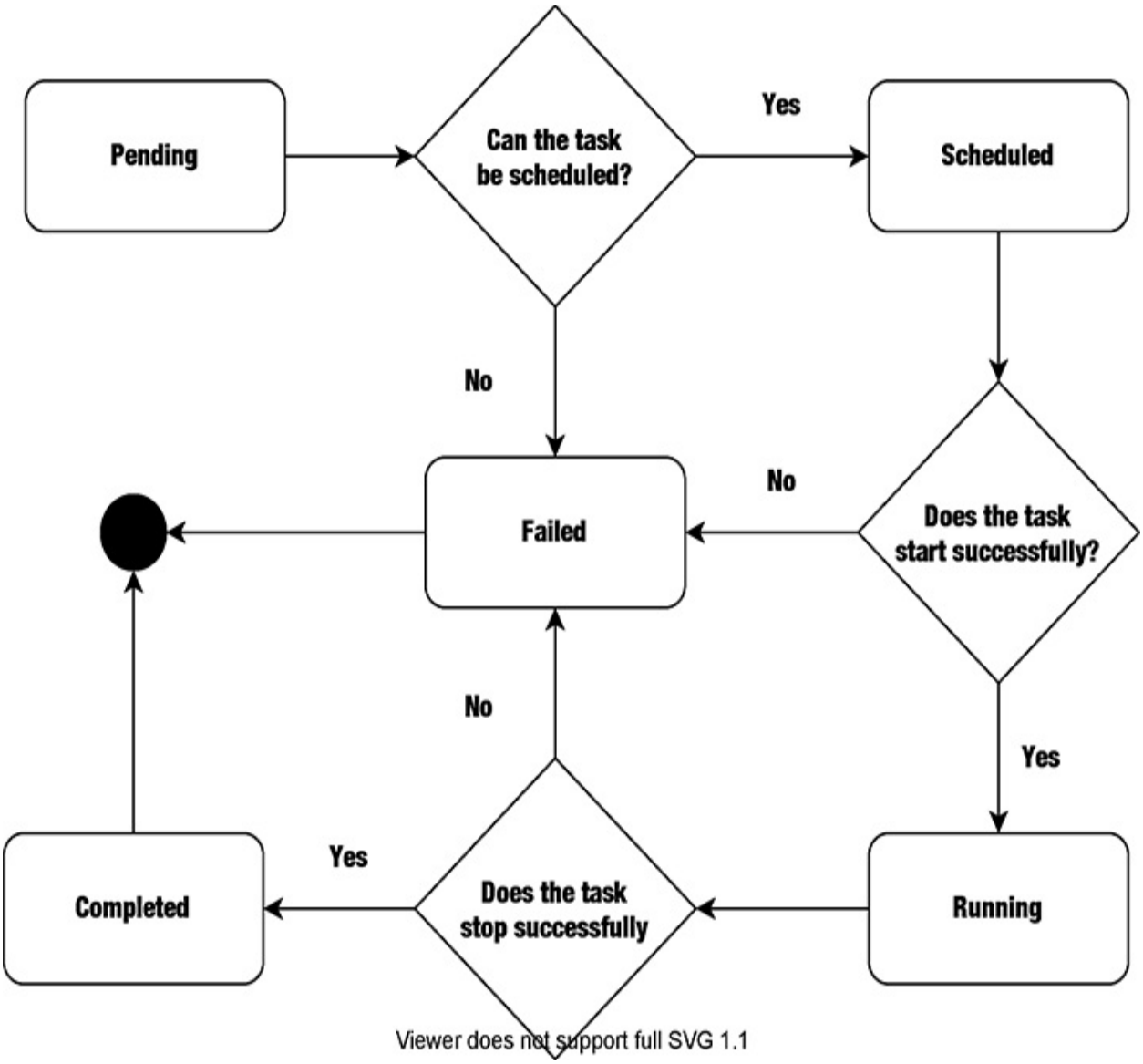
For the rest of the chapter, we'll be creating new files in our project directory. Take the time now to create the following directories and files:

```
.
├── main.go
├── manager
│   └── manager.go
├── node
│   └── node.go
├── scheduler
│   └── scheduler.go
├── task
│   └── task.go
└── worker
    └── worker.go
```

2.1 The task skeleton

The first thing we want to think about is the states a task will go through during its life. First, a user submits a task to the system. At this point, the task has been enqueued but is waiting to be scheduled. Let's call this initial state *Pending*. Once the system has figured out where to run the task, we can say it has been moved into a state of *Scheduled*. The scheduled state means the system has determined there is a machine that can run the task, but it is in the process of sending the task to the selected machine or the selected machine is in the process of starting the task. Next, if the selected machine successfully starts the task, it moves into the *Running* state. Upon a task completing its work successfully, or being stopped by a user, the task moves into a state of *Completed*. If at any point the task crashes or stops working as expected, the task then moves into a state of *Failed*.

Figure 2.2. The states a task will go through during its life cycle.



Now that we have identified the states of a task, lets create the `State` type in Listing 2.1.

Listing 2.1. The State type represents the states a task goes through, from Pending, Scheduled, Running, to Failed or Completed.

```
package task

type State int

const (
        Pending State = iota
        Scheduled
```

```
        Running
        Completed
        Failed
)
```

Next, we should identify other attributes of a task that would be useful for our system. Obviously, an `ID` would allow us to uniquely identify individual tasks, and we'll use universally unique identifiers (UUID) for these. A human-readable `Name` would be good, too, because it means we can talk about `Tim's awesome task` instead of task `74560f1a-b141-40ec-885a-64e4b36b9f9c`. With these, we can sketch the beginning of our Task struct as in Listing 2.2 below.

What is a UUID?

UUID stands for universally unique identifier. A UUID is 128 bits long and, in practice, unique. While it's not improbable that one could generate two identical UUIDs, the probability is extremely low. For more details about UUIDs, see RFC 4122 (https://tools.ietf.org/html/rfc4122).

Listing 2.2. The initial Task struct. Note that the `State` field is of type `State`, which we defined previously.

```
import (
    "github.com/google/uuid"
)

type Task struct {
    ID      uuid.UUID
    Name    string
    State   State
}
```

We have already said we're going to limit our orchestrator to dealing with Docker containers. As a result, we'll want to know what Docker image a task should use, and for that let's use an attribute named `Image`. Given that our tasks will be Docker containers, there are several attributes that would be useful for a task to track. `Memory` and `Disk` will help the system identify the amount of resources a task needs. `ExposedPorts` and `PortBindings` are used by Docker to ensure the machine allocates the proper network ports for the task and that it is available on the network. We'll also want a `RestartPolicy`

attribute, which will tell the system what to do in the event a task stops or fails unexpectedly. With these attributes, we can update our `Task` struct as seen in Listing 2.3.

Listing 2.3. Updating our Task struct with Docker-specific fields.

```
import (
    "github.com/google/uuid"
    "github.com/docker/go-connections/nat"
)

type Task struct {
    ID      uuid.UUID
    Name    string
    State   State
    Image   string
    Memory  int
    Disk    int
    ExposedPorts nat.PortSet
    PortBindings map[string]string
    RestartPolicy string
}
```

Finally, in order to know when a task starts and stops, we can add `StartTime` and `FinishTime` fields to our struct. While these aren't strictly necessary, they are helpful to display in a CLI. With these two attributes, we can flesh out the remainder of our `Task` struct as seen in Listing 2.4.

Listing 2.4. Adding start and stop time fields to the Task struct.

```
type Task struct {
    ID      uuid.UUID
    Name    string
    State   State
    Image   string
    Memory  int
    Disk    int
    ExposedPorts nat.PortSet
    PortBindings map[string]string
    RestartPolicy string
    StartTime time.Time
    FinishTime time.Time
}
```

We have our `Task` struct defined, which represents a task that a user wants to run on our cluster. As we mentioned above, a `Task` can be in one of several states: *Pending*, *Scheduled*, *Running*, *Failed*, or *Completed*. The `Task` struct works fine when a user first requests a task to be run, but how does a user tell the system to stop a task? For this purpose, let's introduce the `TaskEvent` struct seen in Listing 2.5 below.

In order to identify a `TaskEvent`, it will need an `ID`, and like our `Task` this will be done using a UUID. The event will need a `State`, which will indicate the state the task should transition to (e.g. from *Running* to *Completed*). Next, the event will have a `Timestamp` to record the time the event was requested. Finally, the event will contain a `Task` struct. Users won't directly interact with the `TaskEvent` struct. It will be an internal object that our system uses to trigger tasks from one state to another.

Listing 2.5. The TaskEvent struct, which represent an event that moves a Task from one state to another.

```
type TaskEvent struct {
    ID      uuid.UUID
    State State
    Timestamp time.Time
    Task Task
}
```

With our `Task` and `TaskEvent` structs defined, let's move on to sketching the next component, the `Worker`.

2.2 The worker skeleton

If we think of the *task* as the foundation of this orchestration system, then we can think of the *worker* as the next layer that sits atop the foundation. Let's remind ourselves what the worker's requirements are:

1. Run tasks as Docker containers.
2. Accept tasks to run from a manager.
3. Provide relevant statistics to the manager for the purpose of scheduling tasks.
4. Keep track of its tasks and their state.

Using the same process we used for defining the task struct, let's create the `Worker` struct. Given the first and fourth requirements, we know that our worker will need to run and keep track of tasks. To do that, the worker will use a field named `Db`, which will be a map of UUIDs to tasks. To meet the second requirement, accepting tasks from a manager, the worker will want a `Queue` field. Using a queue will ensure that tasks are handled in first-in, first-out (FIFO) order. We won't be implementing our own queue, however, instead opting to using the Queue from `golang-collections`. We'll also add a `TaskCount` field as a convenient way of keeping track of the number of tasks a worker has at any given time.

In your project directory, create a sub-directory called `worker`, and then change into that directory. Now, open a file named `worker.go` and type in the code from Listing 2.6.

Listing 2.6. The beginnings of the Worker struct. Note that by using a map for the Db field, we get the benefit of a datastore without having to worry about the complexities of an external database server or embedded database library.

```
package worker

import (
    "github.com/google/uuid"
    "github.com/golang-collections/collections/queue"

    "cube/task"
)

type Worker struct {
        Name      string
        Queue     queue.Queue
        Db        map[uuid.UUID]task.Task
        TaskCount int
}
```

So, we've identified the fields of our `Worker` struct. Now, let's add some methods that will do the actual work. First, we'll give the struct a `RunTask` method. As its name suggests, it will handle running a task on the machine where the worker is running. Since a task can be in one of several states, the `RunTask` method will be responsible for identifying the task's current state, and then either starting or stopping a task based on the state. Next, let's add a

`StartTask` and a `StopTask` method, which will do exactly as their names suggest—start and stop tasks. Finally, let's give our worker a `CollectStats` method which can be used to periodically collect statistics about the worker.

Listing 2.7. The skeleton of the Worker component. Notice that each method simply prints out a line stating what it will do. Later in the book we will revisit these methods to implement the real behavior represented by these statements.

```
func (w *Worker) CollectStats() {
    fmt.Println("I will collect stats")
}

func (w *Worker) RunTask() {
    fmt.Println("I will start or stop a task")
}

func (w *Worker) StartTask() {
    fmt.Println("I will start a task")
}

func (w *Worker) StopTask() {
    fmt.Println("I will stop a task")
}
```

2.3 The manager skeleton

Along with the `Worker`, the `Manager` is the other major component of our orchestration system. It will handle the bulk of the work.

As a reminder, here are the requirements for the manager we defined in Chapter 1:

1. Accept requests from users to start and stop tasks.
2. Schedule tasks onto worker machines.
3. Keep track of tasks, their states, and the machine on which they run.

In the `manager.go` file, let's create the struct named *Manager* seen in Listing 2.8. The `Manager` will have a queue, represented by the `pending` field, in which tasks will be placed upon first being submitted. The queue will allow the manager to handle tasks on a first-in-first-out (FIFO) basis. Next, the `Manager` will have two in-memory databases: one to store tasks and another

to store task events. The databases are maps of strings to `Task` and `TaskEvent` respectively.

Our `Manager` will need keep track of the workers in the cluster. For this, let's use a field named, surprisingly, `workers`, which will be a slice of strings. Finally, let's add a couple convenience fields that will make our lives easier down the road. It's easy to imagine that we'll want to know the jobs that are assigned to each worker. We'll use a field called `workerTaskMap`, which will be a map of strings to task UUIDs. Similarly, it'd be nice to have an easy way to find the worker running a task given a task name. Here we'll use a field called `taskWorkerMap`, which is a map of task UUIDs to strings, where the string is the name of the worker.

Listing 2.8. The beginnings of our Manager skeleton.

```
package manager


type Manager struct {
    Pending queue.Queue
    TaskDb map[string][]Task
    EventDb map[string][]TaskEvent
    Workers []string
    WorkerTaskMap map[string][]uuid.UUID
    TaskWorkerMap map[uuid.UUID]string
}
```

From our requirements, you can see the manager needs to schedule tasks onto workers. So, let's create a method on our `Manager` struct called `selectWorker` to perform that task. This method will be responsible for looking at the requirements specified in a `Task` and evaluating the resources available in the pool of workers to see which worker is best suited to run the task. Our requirements also say the `Manager` must keep "track of tasks, their states, and the machine on which they run." To meet this requirement, create a method called `UpdateTasks`. Ultimately, this method will end up triggering a call to a worker's `CollectStats` method, but more about later in the book.

Is our `Manager` skeleton missing anything? Ah, yes. So far it can select a worker for a task, and update existing tasks. There is another requirement that is implied in the requirements: the `Manager` obviously needs to send tasks to

workers. So, let’s add this to our requirements and create a method on our `Manager` struct.

Listing 2.9. Like the Worker’s methods, the Manager’s methods only print out what they will do. The work of implementing these methods' actual behavior will come later.

```
func (m *Manager) SelectWorker() {
    fmt.Println("I will select an appropriate worker")
}

func (m *Manager) UpdateTasks() {
    fmt.Println("I will update tasks")
}

func (m *Manager) SendWork() {
    fmt.Println("I will send work to workers")
}
```

2.4 The scheduler skeleton

The last of the four major components from our mental model is the scheduler. Its requirements are as follows:

1. Determine a set of candidate workers on which a task could run.
2. Score the candidate workers from best to worst.
3. Pick the worker with the best score.

This skeleton, which we’ll create in the `scheduler.go` file, will be different from our previous ones. Instead of defining structs and the methods of those structs, we’re going to create an *interface*.

Interfaces in Go

Interfaces are the mechanism by which Go supports polymorphism. They are contracts that specify a set of behaviors, and any type that implements the behaviors can then be used anywhere that the interface type is specified.

For more details about interfaces, see the *Interfaces and other types* section of the *Effective Go* blog post:
https://golang.org/doc/effective_go#interfaces_and_types

Why an interface? As with everything in software engineering, tradeoffs are the norm. During my initial experiments with writing an orchestrator, I wanted a simple scheduler, because I wanted to focus on other core features like running a task. For this purpose, my initial scheduler used a round-robin algorithm that kept a list of workers and identified which worker got the most recent task. Then, when the next task came in, the scheduler simply picked the next worker in its list.

While the round-robin scheduler worked for this particular situation, it obviously has flaws. What happens if the next worker to be assigned a task doesn't have the available resources? Maybe the current tasks are using up all the memory and disk. Furthermore, I might want more flexibility in how tasks are assigned to workers. Maybe I'd want the scheduler to fill up one worker with tasks instead of spreading the tasks across multiple workers, where each worker could potentially only be running a single task. Conversely, maybe I'd want to spread out the tasks across the pool of resources to minimize the likelihood of resource starvation.

Thus, we'll use an interface to specify the methods that a type must implement to be considered a `Scheduler`. As you can see in Listing 2.10, these methods are `SelectCandidateNodes`, `Score`, and `Pick`. Each of these methods map nicely onto the requirements for our scheduler.

Listing 2.10. The skeleton of the Scheduler component.

```
package scheduler

type Scheduler interface {
    SelectCandidateNodes()
    Score()
    Pick()
}
```

2.5 Other skeletons

At this point, we've created skeletons for the four primary objects we see in our mental model: `Task`, `Worker`, `Manager`, and `Scheduler`. There is, however, another object that is hinted at in this model, the *Node*.

Up to now, we've talked about the `Worker`. The `Worker` is the component that deals with our logical workload, that is *tasks*. The `Worker` has a physical aspect to it, however, in that it runs on a physical machine itself, and it also causes tasks to run on a physical machine. Moreover, it needs to know about the underlying machine in order to gather stats about the machine that the manager will use for scheduling decisions. This physical aspect of the `Worker` we'll call a `Node`.

In the context of Cube, a *node* is an object that represents any machine in our cluster. For example, the manager is one type of *node* in Cube. The worker, of which there can be more than one, is another type of *node*. The manager will make extensive use of node objects to represent workers.

For now, we're only going to define the fields that make up a `Node` struct, as seen in Listing 11. First, a node will have a *Name*, for example something as simple as "node-1". Next, a node will have an `Ip` address, which the manager will want to know in order to send tasks to it. A physical machine also has a certain amount of `Memory` and `Disk` space that can be used by tasks. These attributes represent maximum amounts. At any point in time, the tasks on a machine will be using some amount of memory and disk, which we can call `MemoryAllocated` and `DiskAllocated`. Finally, a `Node` will have zero or more tasks, which we can track using a `TaskCount` field.

Listing 2.11. The Node component represents a physical machine where the worker and tasks will run.

```
package node

type Node struct {
        Name            string
        Ip              string
        Cores           int
        Memory          int
        MemoryAllocated int
        Disk            int
        DiskAllocated   int
        Role            string
        TaskCount       int
}
```

2.6 Taking our skeletons for a spin

Now that we've created these skeletons, let's see if we can use them in a simple test program. We want to ensure that the code we just wrote will compile and run. To do this, we're going to create instances of each of the skeletons, print the skeletons, and, finally, call each skeleton's methods.

The following list summarizes in more detail what our test program will do:

- create a `Task` object
- create a `TaskEvent` object
- print the `Task` and `TaskEvent` objects
- create a `Worker` object
- print the `Worker` object
- call the worker's methods
- create a `Manager` object
- call the manager's methods
- create a `Node` object
- print the `Node` object

Before we write this program, however, let's take care of a small administrative task that's necessary to get our code to compile. Remember, we said we're going to use the `queue` implementation from the golang-collections package, and we're also using the `UUID` package from Google. We've also used the `nat` package from Docker. While we have imported them in our code, we haven't yet installed them locally. So, let's do that now:

Listing 2.12. Using the `go get` command to install the third-party packages we imported in our code.

```
$ go get github.com/golang-collections/collections/queue
$ go get github.com/google/uuid
$ go get github.com/docker/go-connections/nat
```

Listing 2.13. Testing the skeletons by creating a minimal program that will actually compile and run.

```
package main
```

```
import (
        "cube/node"
        "cube/task"
        "fmt"
        "time"

        "github.com/golang-collections/collections/queue"
        "github.com/google/uuid"

        "cube/manager"
        "cube/worker"
)

func main() {
        t := task.Task{
                ID:     uuid.New(),
                Name:   "Task-1",
                State:  task.Pending,
                Image:  "Image-1",
                Memory: 1024,
                Disk:   1,
        }

        te := task.TaskEvent{
                ID:        uuid.New(),
                State:     task.Pending,
                Timestamp: time.Now(),
                Task:      t,
        }

        fmt.Printf("task: %v\n", t)
        fmt.Printf("task event: %v\n", te)

        w := worker.Worker{
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]task.Task),
        }
        fmt.Printf("worker: %v\n", w)
        w.CollectStats()
        w.RunTask()
        w.StartTask()
        w.StopTask()

        m := manager.Manager{
                Pending: *queue.New(),
                TaskDb:  make(map[string][]task.Task),
                EventDb: make(map[string][]task.TaskEvent),
                Workers: []string{w.Name},
```

```
        }

        fmt.Printf("manager: %v\n", m)
        m.SelectWorker()
        m.UpdateTasks()
        m.SendWork()

        n := node.Node{
                Name:   "Node-1",
                Ip:     "192.168.1.1",
                Cores:  4,
                Memory: 1024,
                Disk:   25,
                Role:   "worker",
        }

        fmt.Printf("node: %v\n", n)
}
```

Now is the moment of truth! Time to compile and run our program. Do this using the `go run main.go` command, and you should see output like that in Listing 2.14 below.

Listing 2.14. Testing the skeletons by creating a minimal program that will actually compile and run.

```
$ go run main.go #1
task: {389e41e6-95ca-4d48-8211-c2f4aca5127f Task-1 0 Image-1 1024
task event: {69de4b79-9023-4099-9210-d5c0791a2c32 0 2021-04-10 17
worker: { {<nil> <nil> 0} map[] 0} #4
I will collect stats #5
I will start or stop a task
I will start a task
I Will stop a task
manager: {{<nil> <nil> 0} map[] map[] [] map[] map[]} #6
I will select an appropriate worker #7
I will update tasks
I will send work to workers
node: {Node-1 192.168.1.1 4 1024 0 25 0 worker 0} #8
```

Congrats! You've just written the skeleton of an orchestration system that compiles and runs. Take a moment and celebrate. In the following chapters, we'll use these skeletons as a starting point for more detailed discussions of each component before diving into the technical implementations.

2.7 Summary

- Writing skeletons can help translate a mental model from an abstract concept into working code. Thus, we created skeletons for the `Task`, `Worker`, `Manager`, and `Scheduler` components of our orchestration system. This step also helped us identify additional concepts we didn't initially think of. The `TaskEvent` and `Node` components were not represented in our model, but will be useful later in the book.
- A task can be in one of five states: *Pending*, *Scheduled*, *Running*, *Completed*, or *Failed*. The worker and manager will use these states to perform actions on tasks, such as stopping and starting them.
- Go implements *polymorphism* by using *interfaces*. An *interface* is a type that specifies a set of behaviors, and any other type that implements those behaviors will be considered of the same type as the interface. Using an interface will allow us to implement multiple schedulers, each with slightly different behavior.

3 Hanging some flesh on the task skeleton

This chapter covers

- Reviewing how to start and stop Docker containers via the commandline
- Introducing the Docker API calls for starting and stopping containers
- Implementing the Task concept to start and stop a container

Think about cooking your favorite meal. Let's say you like making homemade pizza. In order to end up pulling a delicious, hot pizza out of your oven, you have to perform a number of tasks. If you like onions, green peppers, or any other veggies on your pizza, you have to cut them up. You have to knead the dough into a baking sheet. Next, you spread tomato sauce across the dough and sprinkle cheese over it. Finally, on top of the cheese, you layer your veggies and other ingredients.

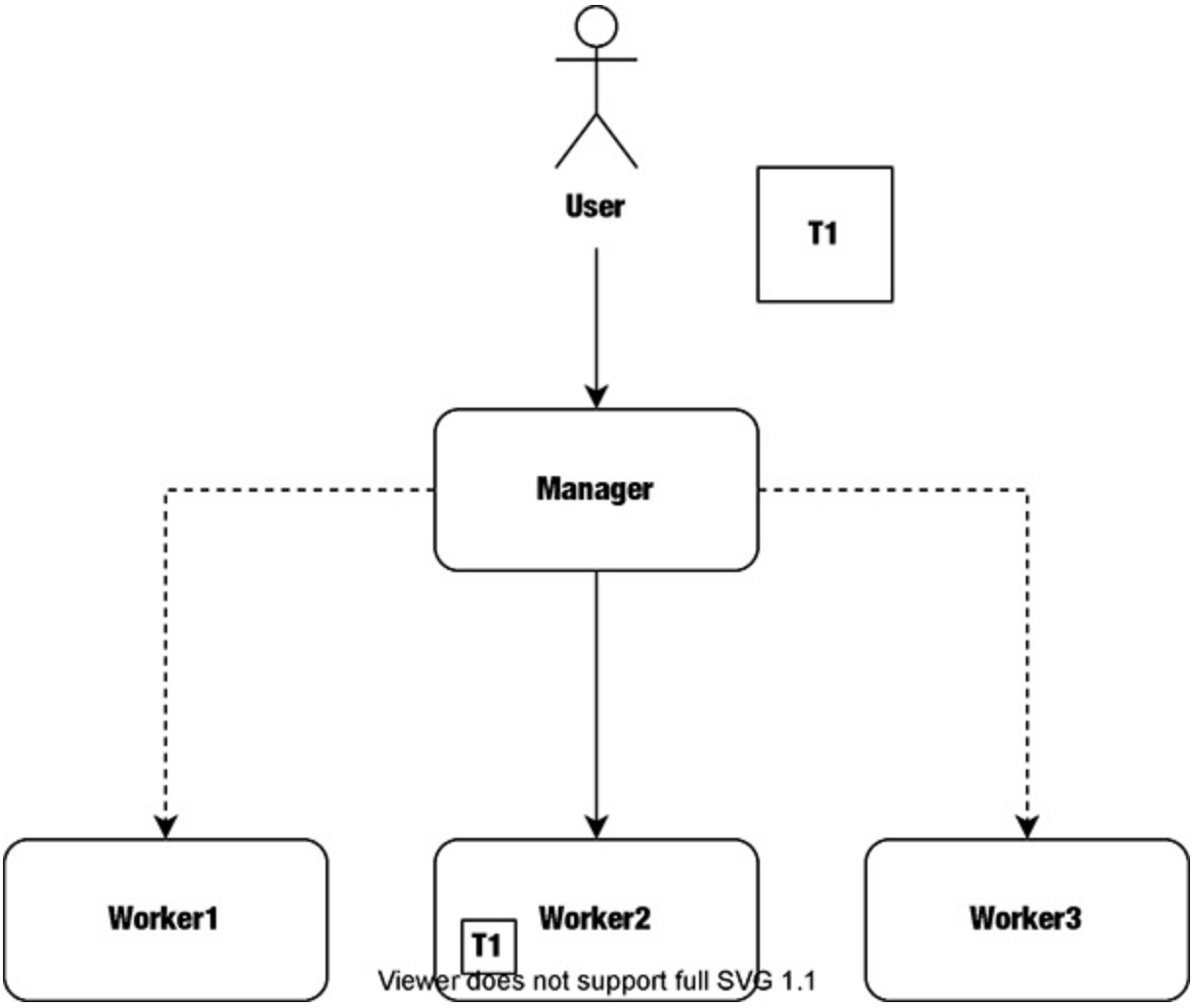
A task in an orchestration system is similar to one of the individual steps in making a pizza. Like most companies these days, yours most likely has a website. That company's website runs on a web server, perhaps the ubiquitous Apache web server. That's a task. The website may use a database, like MySQL or PostgreSQL, to store dynamic content. That's a task.

In our pizza making analogy above, the pizza wasn't made in a vacuum. It was created in a specific context, which is a kitchen. The kitchen provides the necessary resources to make the pizza: there is a refrigerator where the cheese is stored; cabinets where the pizza sauce is kept; an oven in which to cook the pizza; knives to cut the pizza into slices.

Similarly, a task operates in a specific context. In our case, that context will be a Docker container. Like the kitchen, the container will provide the resources necessary for the task to run: it will provide CPU cycles, memory, and networking according to the needs of the task.

As a reminder, the task is the foundation of an orchestration system. Figure 1.1 shows a modified version of our mental model from chapter 1.

Figure 3.1. The main purpose of an orchestration system is to accept tasks from users and run them on the system's worker nodes. Here, we see a user submitting a task to the Manager node, which then selects Worker2 to run the task. The dotted lines to Worker1 and Worker3 represent that these nodes were considered but ultimately not select to run the task.



In the rest of this chapter, we'll flesh out the `Task` skeleton we wrote in the previous chapter. But first, let's quickly review some Docker basics.

3.1 Docker: starting, stopping, and inspecting containers from the commandline

If you are a developer, you have probably used Docker containers to run your application and its backend database on your laptop while working on your code. If you are a DevOps engineer, you may have deployed Docker containers to your company's production environment. Containers allow the developer to package their code, along with all its dependences, and then ship the container to production. If a DevOps team is responsible for deployments to production, then they only have to worry about deploying the container. They don't have to worry about whether the machine where the container will run has the correct version of the PostgreSQL library that the application uses to connect to its database.

If you need a more detailed review of Docker containers and how to control them, check out chapter 2 of Docker In Action.

To run a Docker container, we can use the `docker run` command, an example of which can be seen in Listing 3.1. Here, the `docker run` command is starting up a PostgreSQL database in a container, which might be used as a backend datastore while developing a new application.

Listing 3.1. Running the Postgres database server as a Docker container. This command will run the container in the foreground, meaning we can see its log output (`-it`), gives the container a name of `postgres`, sets the POSTGRES_USER and POSTGRES_PASSWORD environment variables, and uses version 13 of the Postgres server.

```
$ docker run -it \
    -p 5432:5432 \
    --name postgres \
    -e POSTGRES_USER=cube \
    -e POSTGRES_PASSWORD=secret \
    postgres:13
```

Once a container is running, then it performs the same functions it would if you were running it as a regular process on your laptop or desktop. In the case of the Postgres database from Listing 3.1, I can now log into the database server using the `psql` commandline client and create a table like that in Listing 3.2.

Listing 3.2. Logging in to the Postgres server and creating a table. Because we specified `-p 5432:5432` in the `docker run` command in the previous listing, we can tell the `psql` client to connect to that port on the local machine.

```
$ psql -h localhost -p 5432 -U cube
Password for user cube:
psql (9.6.22, server 13.2 (Debian 13.2-1.pgdg100+1))
WARNING: psql major version 9.6, server major version 13.
         Some psql features might not work.
Type "help" for help.

cube=# \d
No relations found.
cube=# CREATE TABLE book (
isbn char(13) PRIMARY KEY,
title varchar(240) NOT NULL,
author varchar(140)
);
CREATE TABLE
cube=# \d
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | book | table | cube
(1 row)
```

Once a container is up and running, we can get information about it by using the `docker inspect` command. The output from this command is extensive, so I will only list the `State` info.

Listing 3.3. Using the `docker inspect cube-book` command to get information about the running container.

```
$ docker inspect cube-book
[
    {
        "Id": "a820c7abb54b723b5efc0946900baf58e093d8fdd238d4ec7c
        "Created": "2021-05-15T20:00:41.228528102Z",
        "Path": "docker-entrypoint.sh",
        "Args": [
            "postgres"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 27599,
```

```
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-05-15T20:00:42.4656334Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        ....
]
```

Finally, we can stop a Docker container using the `docker stop cube-book` command. There isn't any output from the command, but if we run the `docker inspect cube-book` command now, we'll see the state has changed from `running` to `exited`.

Listing 3.4. Running `docker inspect cube-book` after `docker stop cube-book`.

```
$ docker inspect cube-book
[
    {
        "Id": "a820c7abb54b723b5efc0946900baf58e093d8fdd238d4ec7c
        "Created": "2021-05-15T20:00:41.228528102Z",
        "Path": "docker-entrypoint.sh",
        "Args": [
            "postgres"
        ],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-05-15T20:00:42.4656334Z",
            "FinishedAt": "2021-05-15T20:18:31.698919838Z"
        },
        ....
]
```

3.2 Docker: starting, stopping, and inspecting containers from the API

In our orchestration system, the worker will be responsible for starting, stopping, and providing information about the tasks its running. To perform these functions, the worker will use Docker's API. The API is accessible via the HTTP protocol using a client like `curl` or the HTTP library of a programming language. Listing 3.5 shows an example of using `curl` to get the same information we got from the `docker inspect` command previously.

Listing 3.5. Querying the Docker API with the `curl` HTTP client. Notice we're passing the `--unix-socket` flag to the curl command. By default, Docker listens on a `unix` socket, but it can be configured to listen on a `tcp` socket. The URL, `http://docker/containers/6970e8469684/json`, contains the ID of the container to inspect, which I got from the `docker ps` command on my machine. Finally, the output from curl is piped to the `jq` command, which prints the output in a more readable format than curl's.

```
curl --unix-socket \
    /var/run/docker.sock http://docker/containers/6970e8469684/js
{
  "Id": "6970e8469684d439c73577c4caee7261bf887a67433420e7dcd637cc
  "Created": "2021-05-15T20:58:36.283909602Z",
  "Path": "docker-entrypoint.sh",
  "Args": [
    "postgres"
  ],
  "State": {
    "Status": "running",
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 270523,
    "ExitCode": 0,
    "Error": "",
    "StartedAt": "2021-05-15T20:58:36.541148947Z",
    "FinishedAt": "0001-01-01T00:00:00Z"
  },
  ....
}
```

We could use Go's HTTP library in our orchestration system, but that would force us to deal with many low-level details like HTTP methods, status codes, and serializing requests and deserializing responses. Instead, we're going to use Docker's SDK, which handles all the low-level HTTP details for us and allows us to focus on our primary task: creating, running, and stopping

containers. The SDK provides the following six methods that will meet our needs:

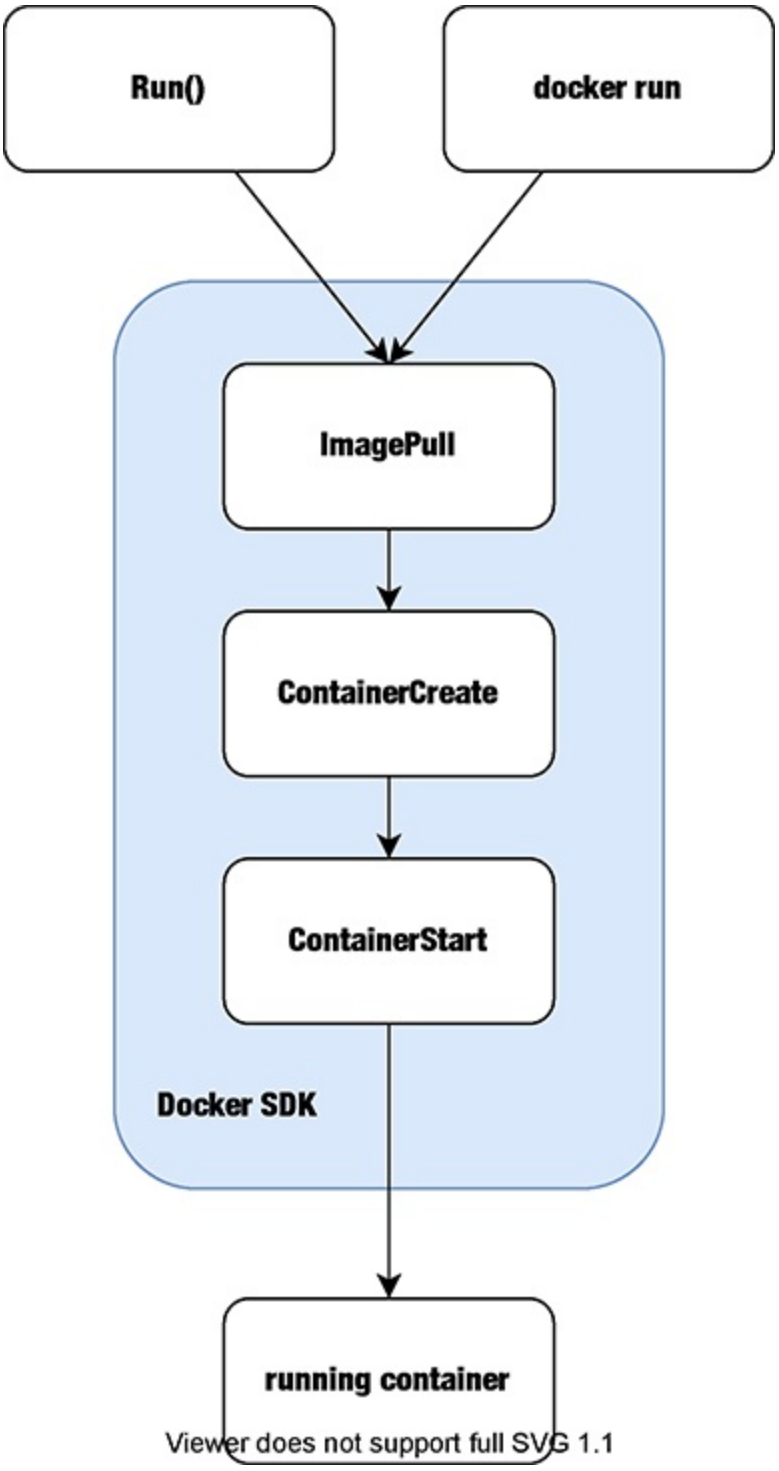
- `NewClientWithOpts`: a helper method that instantiates an instance of the client and returns it to the caller
- `ImagePull`: pulls the image down to the local machine where it will be run
- `ContainerCreate`: creates a new container with a given configuration
- `ContainerStart`: sends a request to Docker Engine to start the newly created container
- `ContainerStop`: sends a request to Docker Engine to stop a running container
- `ContainerRemove`: removes the container from the host

 **Note**

Docker's Golang SDK has extensive documentation (https://pkg.go.dev/github.com/docker/docker) that's worth reading. In particular, the docs about the Go client (https://pkg.go.dev/github.com/docker/docker/client) are relevant to our work throughout the rest of this book.

The `docker` commandline examples we reviewed in the previous section actually use the Go SDK under the hood. Later in this chapter, we'll implement a `Run()` method that uses the `ImagePull`, `ContainerCreate`, and `ContainerStart` methods to create and start a container. Figure 3.1 provides a graphic representation of our custom code and the `docker` command using the SDK.

Figure 3.2. Regardless of the starting point, all paths to creating and running a container go through the Docker SDK.



By using the Go SDK for controlling the Docker containers in our orchestration system, we don’t have to reinvent the wheel. We can simply reuse the same code used by the `docker` command every day.

3.3 Task configuration

In order to run our tasks as containers, they need a configuration. What is a configuration. Think back to our pizza analogy from the beginning of the chapter. One of the tasks in making our pizza was cutting the onions (if you don’t like onions, insert your veggie of choice). To perform that task, we would use a knife and a cutting board, and we would cut the onions in a particular way. Perhaps we cut them into thin, even slices, or dice them in small cubes. This is all part of the "configuration" of the task of cutting onions. (Okay, I’m probably stretching the pizza analogy a bit far, but I think you get the point.)

For a task in our orchestration system, we’ll describe its configuration using the `Config` struct in Listing 3.6. This struct encapsulates all the necessary bits of information about a task’s configuration. The comments should make the intent of each field obvious, but there are a couple fields worth highlighting.

The `Name` field will be used to identify a task in our orchestration system, and it will perform double duty as the name of the running container. Throughout the rest of the book, we’ll use this field to name our containers like "test-container-1".

The `Image` field, as you probably guessed, holds the name of the image the container will run. Remember, an image can be thought of as a package: it contains the collection of files and instructions necessary to run a program. This field can be set to a value as simple as `postgres`, or it can be set to a more specific value that includes a version, like `postgres:13`.

The `Memory` and `Disk` fields will serve two purposes. The scheduler will use them to find a node in the cluster capable of running a task. They will also be used to tell the Docker daemon the amount of resources a task requires.

The `Env` field allows a user to specify environment variables that will get passed in to the container. In our command to run a Postgres container we set two environment variables: `-e POSTGRES_USER=cube` to specify the database user and `-e POSTGRES_PASSWORD=secret` to specify that user’s password.

Finally, the `RestartPolicy` field tells the Docker daemon what to do in the event a container dies unexpectedly. This field is one of the mechanisms that provides resilience in our orchestration system. As you can see from the comment, the acceptable values are an empty string, `always`, `unless-stopped`, or `on-failure`. Setting this field to `always` will, as its name implies, restart a container if it stops. Setting it to `unless-stopped` will restart a container unless it has been stopped (e.g. by `docker stop`). Setting it to `on-failure` will restart the container if it exits due to an error (i.e. a non-zero exit code). There are a few details that are spelled out in the documentation (https://docs.docker.com/config/containers/start-containers-automatically/#restart-policy-details)

We're going to add the `Config` struct in listing 3.6 to the `task.go` file from chapter 2.

Listing 3.6. The config struct that will hold the configuration for orchestration tasks.

```
type Config struct {
        Name string
        AttachStdin bool
        AttachStdout bool
        AttachStderr bool
        Cmd []string
        Image string
        Memory int64
    Disk int64
        Env []string
        RestartPolicy string
}
```

3.4 Starting and Stopping Tasks

Now that we've talked about a task's configuration, let's move on actually starting and stopping a task. Remember, the worker in our orchestration system will be responsible for running tasks for us. That responsibility will mostly involve starting and stopping tasks.

Let's start by adding the code for the `Docker` struct you see in Listing 3.7 to the `task.go` file. This struct will encapsulate everything we need to run our task as a Docker container. The `Client` field will hold a Docker client object

that we'll use to interact with the Docker API. The `Config` field will hold the task's configuration. And, once a task is running, it will also contain the `ContainerId`. This ID will allow us to interact with the running task.

Listing 3.7. Starting and stopping containers.

```
type Docker struct {
    Client *client.Client
        Config  Config
        ContainerId string
}
```

For the sake of convenience, let's create a struct called `DockerResult`. We can use this struct as the return value in methods that start and stop containers, providing a wrapper around common information that is useful for callers. The struct contains an `Error` field to hold any error messages. It has an `Action` field that can be used to identify the action being taken, for example "start" or "stop". It has a `ContainerId` field to identify the container to which the result pertains. And, finally, there is a `Result` field that can hold arbitrary text that provides more information about the result of the operation.

Listing 3.8. The `DockerResult` struct.

```
type DockerResult struct {
        Error       error
        Action      string
        ContainerId string
        Result      string
}
```

Now we're reading for the exciting part: actually writing the code to create and run a task as a container. To do this, let's start by adding a method to the `Docker` struct we created earlier. Let's call that method `Run`.

The first part of our `Run` method will pull the Docker image our task will use from a container registry such as Docker Hub. A container registry is simply a repository of images and allow for the easy distribution of the images it hosts. To pull the image, the `Run` method first creates a context, which is a type that holds values that can be passed across boundaries such as APIs and processes. It's common to use a context to pass along deadlines or

cancellation signals in requests to an API. In our case, we'll use an empty context returned from the `Background` function.

Next, `Run` calls the `ImagePull` method on the Docker client object, passing the context object, the image name, and any options necessary to pull the image. The `ImagePull` method returns two values: an object that fulfills the `io.ReadCloser` interface and an error object. It stores these values in the `reader` and `err` variables.

The next step in the method checks the error value returned from `ImagePull`. If the value is not `nil`, the method prints the error message and returns as a `DockerResult`.

Finally, the method copies the value of the `reader` variable to `os.Stdout` via the `io.Copy` function. `io.Copy` is a function from the `io` package in Golang's standard library, and it simply copies data to a destination (`os.Stdout`) from a source (`reader`). Because we'll be working from the commandline whenever we're running the components of our orchestration system, it's useful to write the `reader` variable to Stdout as a way to communicate what happened in the `ImagePull` method.

Listing 3.9. The start of our `Run()` method. Similar to running a container from the commandline, the method begins by pulling the container's image.

```
func (d *Docker) Run() DockerResult {
        ctx := context.Background()
        reader, err := d.Client.ImagePull(
        ctx, d.Config.Image, types.ImagePullOptions{})
        if err != nil {
                log.Printf("Error pulling image %s: %v\n", d.Conf
                return DockerResult{Error: err}
        }
        io.Copy(os.Stdout, reader)
}
```

Once the `Run` method has pulled the image and checked for errors (and finding none, we hope), the next bit of business on the agenda is to prepare the configuration to be sent to Docker. Before we do that, however, let's take a look at the signature of the `ContainerCreate` method from the Docker client. This is the method we'll use to actually create the container. As you

can see in listing 3.10, `ContainerCreate` takes several arguments. Similar to the `ImagePull` method used earlier, it takes a `context.Context` as its first argument. The next argument is the actual container configuration, which is a pointer to a `container.Config` type. We'll copy the values from our own `Config` type into this one. The third argument is a pointer to a `container.HostConfig` type. This type will hold the configuration a task requires of the host on which the container will run, for example a Linux machine. The fourth argument is also a pointer and points to a `network.NetworkingConfig` type. This type can be used to specify networking details, such as the network ID container will use, any links to other containers that are needed, and IP addresses. For our purposes, we won't make use of the network configuration, instead allowing Docker to handle those details for us. The fifth argument is another pointer, and it points to a `specs.Platform` type. This type can be used to specify details about the platform on which the image runs. It allows you to specify things like the CPU architecture and the operating system. We won't be making use of this argument either. The sixth and final argument to `ContainerCreate` is the container name, passed as a string.

Listing 3.10. The Docker client's `ContainerCreate` method creates a new container based on a configuration.

```
func (cli *Client) ContainerCreate(
    ctx context.Context,
    config *container.Config,
    hostConfig *container.HostConfig,
    networkingConfig *network.NetworkingConfig,
    platform *specs.Platform,
    containerName string) (container.ContainerCreateCreatedBody,
```

Now we know what information we need to pass along in the `ContainerCreate` method, so let's gather it from our `Config` type and massage it into the appropriate types that `ContainerCreate` will accept. What we'll end up with is what you see in listing 3.11.

First, we'll create a variable called `rp`. This variable will hold a `container.RestartPolicy` type, and it will contain the `RestartPolicy` we defined in our `Config` struct from listing 3.6 earlier.

Following the `rp` variable let's declare a variable called `r`. This variable will hold the resources required by the container in a `container.Resources` type. The most common resources we'll use for our orchestration system will be memory.

Next, let's create a variable called `cc` to hold our container configuration. This variable will be of the type `container.Config`, and into it we'll copy two values from our `Config` type. The first is the `Image` our container will use. The second is is any environment variables, which go into the `Env` field.

Finally, we take the `rp` and `r` variables we defined and add them to a third variable called `hc`. This variable is a `container.HostConfig` type. In addition to specifying the `RestartPolicy` and `Resources` in the `hc` variable, we'll also set the `PublishAllPorts` field to true. What does this field do? Remember our example `docker run` command in listing 3.2, where we start up a PostgreSQL container? In that command, we used `-p 5432:5432` to tell Docker that we wanted to map port 5432 on the host running our container to port 5432 inside the container. Well, that's not the best way to expose a container's ports on a host. There is an easier way. Instead, we can set `PublishAllPorts` to true, and Docker will expose those ports automatically by randomly choosing available ports on the host.

Listing 3.11. The next phase of running a container creates four variables to hold configuration information that gets passed to the `ContainerCreate` method.

```
func (d *Docker) Run() DockerResult {

    // previous code not listed

        rp := container.RestartPolicy{
                Name: d.Config.RestartPolicy,
        }

        r := container.Resources{
                Memory: d.Config.Memory,
        }

    cc := container.Config{
        Image: d.Config.Image,
        Env: d.Config.Env,
    }
```

```
        hc := container.HostConfig{
                RestartPolicy: rp,
                Resources:     r,
        PublishAllPorts: true,
        }
```

So we've done all the necessary prep work, and now we can create the container and start it. We've already touched on the `ContainerCreate` method above in listing 3.10, so all that's left to do is to call it like in listing 3.12. One thing to notice, however, is that we pass `nil` values as the fourth and fifth arguments, which you'll recall from listing 3.10 is the `networking` and `platform` arguments. We won't be making use of these features in our orchestration system, so we can ignore them for now.

As with the `ImagePull` method earlier, `ContainerCreate` returns two values, a response, which is a pointer to a `container.ContainerCreateCreatedBody` type, and an error type. The `ContainerCreateCreatedBody` type gets stored in the `resp` variable, and we put the error in the `err` variable. Next, we check the `err` variable for any errors, and if we find any print them and return them in a `DockerResult` type.

Great! We've got all our ingredients together, and we've formed them into a container. All that's left to do is start it. To perform this final step, we call the `ContainerStart` method.

Besides a context argument, `ContainerStart` takes the `ID` of an existing container, which we get from the `resp` variable returned from `ContainerCreate`, and any options necessary to start the container. In our case, we don't need any options, so we simply pass an empty `types.ContainerStartOptions`. `ContainerStart` only returns one type, an error, so we check it in the same we have with the other method calls we've made. If there is an error, we print it and then return it in a `DockerResult`.

Listing 3.12. The penultimate phase of the process calls the `ContainerCreate` and `ContainerStart` methods.

```
func (d *Docker) Run() DockerResult {

    // previous code not listed
```

```
        resp, err := d.Client.ContainerCreate(
        ctx, &cc, &hc, nil, nil, d.Config.Name)
        if err != nil {
                log.Printf(
            "Error creating container using image %s: %v\n",
            d.Config.Image, err
        )
                return DockerResult{Error: err}
        }

        err := d.Client.ContainerStart(
        ctx, resp.ID, types.ContainerStartOptions{})
    if err != nil {
                log.Printf("Error starting container %s: %v\n", r
                return DockerResult{Error: err}
        }
```

At this point, if all was successful, we have a container running the task. All that's left to do now is to take care of some bookkeeping, which you can see in listing 3.13. We start by adding the container ID to the configuration object (which will ultimately be stored, but let's not get ahead of ourselves!). Similar to printing the results of the `ImagePull` operation to stdout, we do the same with the result of starting the container. This is accomplished by calling the `ContainerLogs` method and then writing the return value to stdout using the `stdcopy.StdCopy(os.Stdout, os.Stderr, out)` call.

Listing 3.13. The final phase of creating and running a container involves some bookkeeping and outputting information from logs.

```
func (d *Docker) Run() DockerResult {

    // previous code not listed

        d.Config.Runtime.ContainerID = resp.ID

        out, err := cli.ContainerLogs(
        ctx,
        resp.ID,
        types.ContainerLogsOptions{ShowStdout: true, ShowStderr:
    )
        if err != nil {
                log.Printf("Error getting logs for container %s: 
                return DockerResult{Error: err}
```

```
        }

        stdcopy.StdCopy(os.Stdout, os.Stderr, out)

        return DockerResult{
        ContainerId: resp.ID,
        Action: "start",
        Result: "success"
    }
}
```

As a reminder, the `Run` method we've written in listings 3.9, 3.11, 3.12, and 3.13 perform the same operations as the `docker run` command. When you type `docker run` on the commandline, under the hood the `docker` binary is using the same SDK methods we're using in our code to create and run the container.

Now that we can create a container and start it, let's write the code to stop a container. Compared to our `Run` method above, the `Stop` method will be much simpler, as you can see in Listing 3.10. Because there isn't the necessary prep work to do for stopping a container, the process simply involves calling the `ContainerStop` method with the `ContainerID`, and then calling the `ContainerRemove` method with the `ContainerID` and the requisite options. Again, in both of these operations, the code checks the value of the `err` returned from the method.

As with the `Run` method, our `Stop` method performs the same operations carried out by the `docker stop` and `docker rm` commands.

Listing 3.14. Similar to running a container, stopping a container is a two-step process. First, the container is stopped by calling the `ContainerStop` method, and finally it's removed by calling `ContainerRemove`.

```
func (d *Docker) Stop(id string) DockerResult {
        log.Printf("Attempting to stop container %v", id)
        ctx := context.Background()
        err := d.Client.ContainerStop(ctx, id, nil)
        if err != nil {
                fmt.Println(err)
                panic(err)
        }
```

```
        err = d.Client.ContainerRemove(ctx, id, types.ContainerRe
        if err != nil {
                panic(err)
        }

        return DockerResult{Action: "stop", Result: "success", Er
}
```

Now, let's update our `main.go` program that we created in chapter 2 to create and stop a container.

First, add the `createContainer` function in listing 3.14 to the bottom of the main.go file. Inside it, we'll set up the configuration for the task and store it in a variable called `c`, then we'll create a new Docker client and store it in `dc`. Next, let's create the `d` object, which is of type `task.Docker`. From this object, we call the `Run()` method to create the container.

Listing 3.15. In the `createContainer` function, we use the `Config` and `Docker` objects we wrote earlier in the chapter.

```
func createContainer() (*task.Docker, *task.DockerResult) {
        c := task.Config{
                Name:  "test-container-1",
                Image: "postgres:13",
                Env: []string{
                        "POSTGRES_USER=cube",
                        "POSTGRES_PASSWORD=secret",
                },
        }

        dc, _ := client.NewClientWithOpts(client.FromEnv)
        d := task.Docker{
                Client: dc,
                Config: c,
        }

        result := d.Run()
        if result.Error != nil {
                fmt.Printf("%v\n", result.Error)
                return nil, nil
        }

        fmt.Printf(
        "Container %s is running with config %v\n", result.Contai
        return &d, &result
```

```
}
```

Second, add the `stopContainer` function below `createContainer`. This function accepts a single argument, `d`, which is the same `d` object created in `CreateContainer` in listing 3.14. All that's left to do is call `d.Stop()`.

Listing 3.16. The `stopContainer` function uses the `Docker` object returned from `createContainer` to stop a container.

```
func stopContainer(d *task.Docker) *task.DockerResult {
        result := d.Stop()
        if result.Error != nil {
                fmt.Printf("%v\n", result.Error)
                return nil
        }

        fmt.Printf(
        "Container %s has been stopped and removed\n", result.Con
        return &result
}
```

Finally, we call the `createContainer` and `stopContainer` functions we created above from our `main()` function in main.go. To do that, add the code from Listing 3.15 to the bottom of your main function.

As you can see, the code is fairly simple. It starts by prints a useful message that it's going to create a container, then calls the `createContainer()` function and stored the results in two variables, `dockerTask` and `createResult`. Then, it checks for errors by comparing the value of `createResult.Error` to `nil`. If it finds an error, it prints it and exits by calling `os.Exit(1)`. To stop the container, the main function simply calls `stopContainer` and passes it the `dockerTask` object returned by the earlier call to `createContainer`.

Listing 3.17. Calling the createContainer and stopContainer functions that we created in the previous two listings.

```
func main() {

    // previous code not shown

    fmt.Printf("create a test container\n")
```

```
    dockerTask, createResult := createContainer()
    if createResult.Error != nil {
        fmt.Printf(createResult.Error)
        os.Exit(1)
    }

    time.Sleep(time.Second * 5)
    fmt.Printf("stopping container %s\n", createResult.ContainerI
    _ = stop_container(dockerTask)
}
```

Time for another moment of truth. Let's run the code!

Listing 3.18. Running the code to create and stop a container.

```
$ go run main.go #1
task: {2c66f7c4-2484-4cf8-a22b-81c3dd24294d Task-1 0 Image-1 1024
task event: {f7045213-732e-49f9-9ca0-ef781e58d30c 0 2021-05-16 16
worker: { {<nil> <nil> 0} map[] 0}
I will collect stats
I will start or stop a task
I will start a task
I Will stop a task
manager: {{<nil> <nil> 0} map[] map[] [] map[] map[]}
I will select an appropriate worker
I will update tasks
I will send work to workers
node: {Node-1 192.168.1.1 4 1024 0 25 0 worker 0} #2
create a test container #3
{"status":"Pulling from library/postgres","id":"13"}
{"status":"Digest: sha256:117c3ea384ce21421541515edfb11f2997b2c85
{"status":"Status: Image is up to date for postgres:13"} #4
Container 20dfabd6e7f7f30948690c4352979cbc2122d90b0a22567f9f0bcbc
stopping container 20dfabd6e7f7f30948690c4352979cbc2122d90b0a2256
2021/05/16 16:00:47 Attempting to stop container 20dfabd6e7f7f309
Container 20dfabd6e7f7f30948690c4352979cbc2122d90b0a22567f9f0bcbc
```

At this point, we have the foundation of our orchestration system in place. We can create, run, stop, and remove containers, which provide the technical implementation of our Task concept. The other components in our system, namely the `Worker` and `Manager`, will use this Task implementation to perform their necessary roles.

3.5 Summary

- The task concept, and its technical implementation, is the fundamental unit of our orchestration system. All the other components—worker, manager, and scheduler—exist for the purpose of starting, stopping, and inspecting tasks.
- The Docker API provides the ability to manipulate containers programmatically. The three most important methods are `ContainerCreate`, `ContainerStart`, and `ContainerStop`. These methods allow a developer to perform the same operations from their code that they can do from the commandline, i.e. docker run, docker start, and docker stop.
- A container has a configuration. The configuration can be broken down into the following categories: identification (i.e. how to identify containers), resource allocation, networking, and error handling.
- A task is the smallest unit of work performed by our orchestration system and can be thought of similarly to running a program on your laptop or desktop.
- We use Docker in this book because it abstracts away many of the concerns of the underlying operating system. We could implement our orchestration system to run tasks as regular operating system processes. Doing so, however, means our system would need to be intimately familiar with the details of how process run across OSes (e.g. Linux, Mac, Windows).
- An orchestration system consists of multiple machines, which are called a cluster.

4 Workers of the Cube, unite!

This chapter covers

- Reviewing the purpose of the worker component in an orchestration system
- Reviewing the `Task` and `Docker` structs
- Defining and implementing an algorithm for processing incoming tasks
- Building a simplistic state machine to transition tasks between states
- Implementing the worker's methods for starting and stopping tasks

Think about running a web server that serves static pages. In many cases, running a single instance of our web server on a single physical or virtual machine is good enough. As the site grows in popularity, however, this setup poses several problems:

- Resource availability: given the other processes running on the machine, is there enough memory, CPU, and disk to meet the needs of our web server?
- Resilience: if the machine running the web server goes down, our site goes down with it.

Running multiple instances of our web server helps us solve these problems.

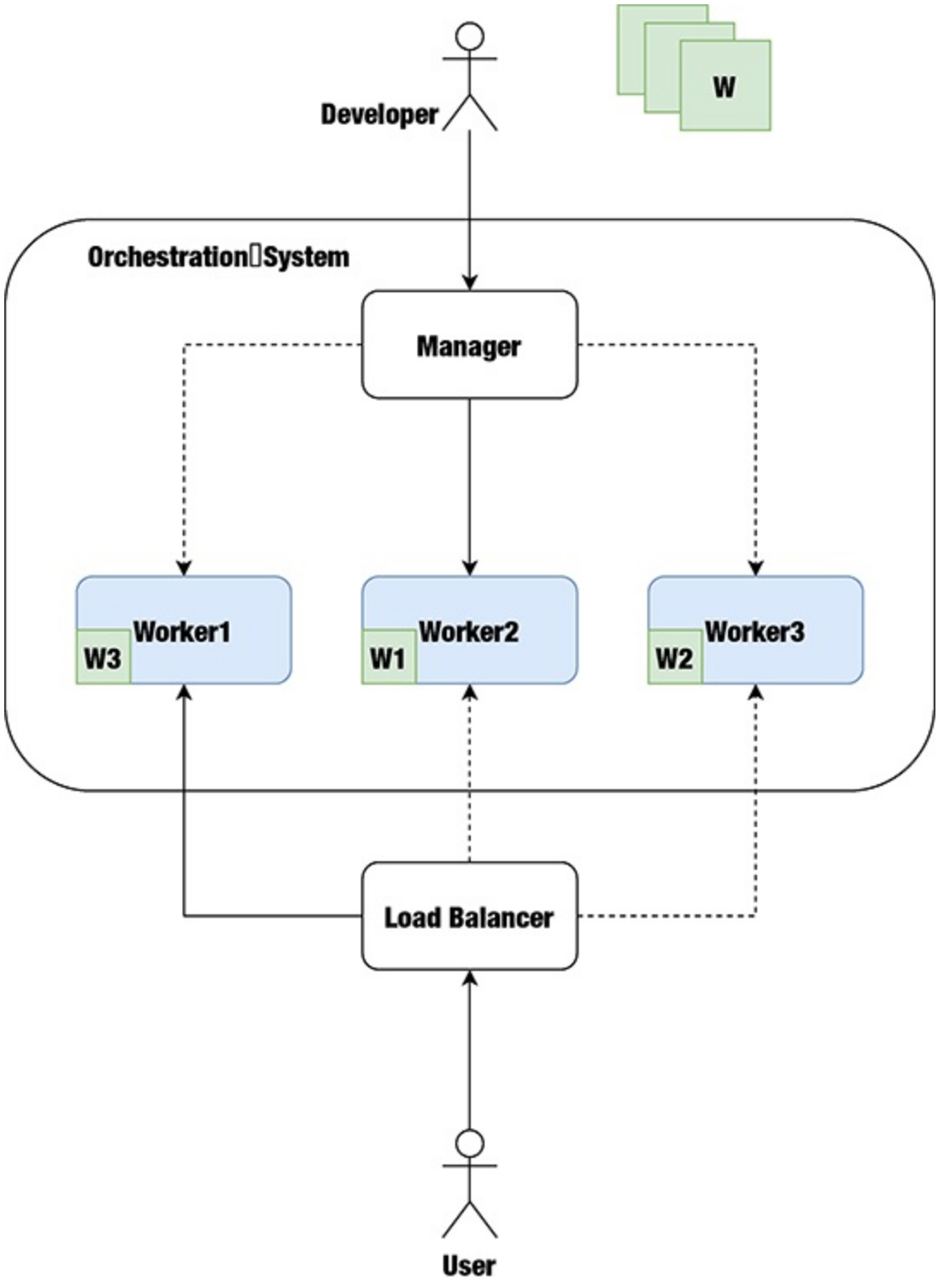
In this chapter we will focus on fleshing out the `Worker` skeleton sketched out in chapter 2. It will use the `Task` implementation we covered in chapter 3. At the end of the chapter, we'll use our implementation of the worker to run multiple instances of a simple web server like that in our scenario above.

4.1 The Cube worker

With an orchestration system, the worker component allows us to easily scale applications such as our web server from the above scenario. Figure 4.1 shows how we could run three instances of our website, represented by the boxes `W1`, `W2`, and `W3`, with each instance running on a separate worker. In this

diagram, it’s important to realize that the term `Worker` is doing double duty: it represents a physical or virtual machine, and the worker component of the orchestration system that runs on that machine.

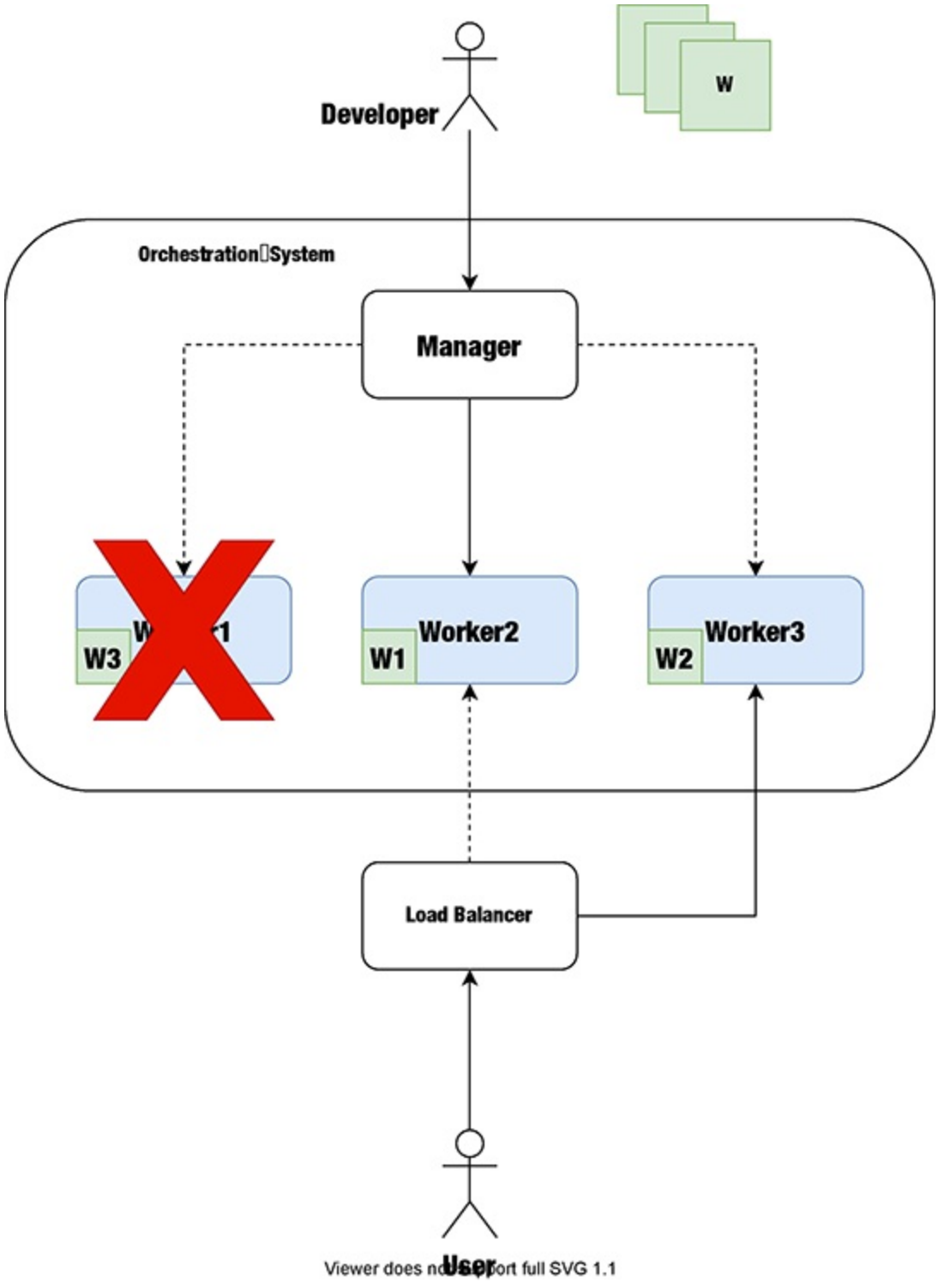
Figure 4.1. The worker boxes are serving double duty in this diagram. They represent a physical or virtual machine, on which the `Worker` component of the orchestration system runs.



Now, we're less likely to experience resource availability issues. Because we're running three instances of our site, on three different workers, user requests can be spread across the three instances instead of going to a single instance running on a single machine.

Similarly, our site is now more resilient to failures. For example, if `Worker1` in figure 4.2 crashes, it will take the `W3` instance of our site with it. While this might make us sad and cause us some work to bring `Worker1` back online, the users of our site shouldn't notice the difference. They'll be able to continue making requests to our site and getting back the expected static content.

Figure 4.2. In the scenario where a worker node fails, our web server running on the other nodes can still respond to requests.

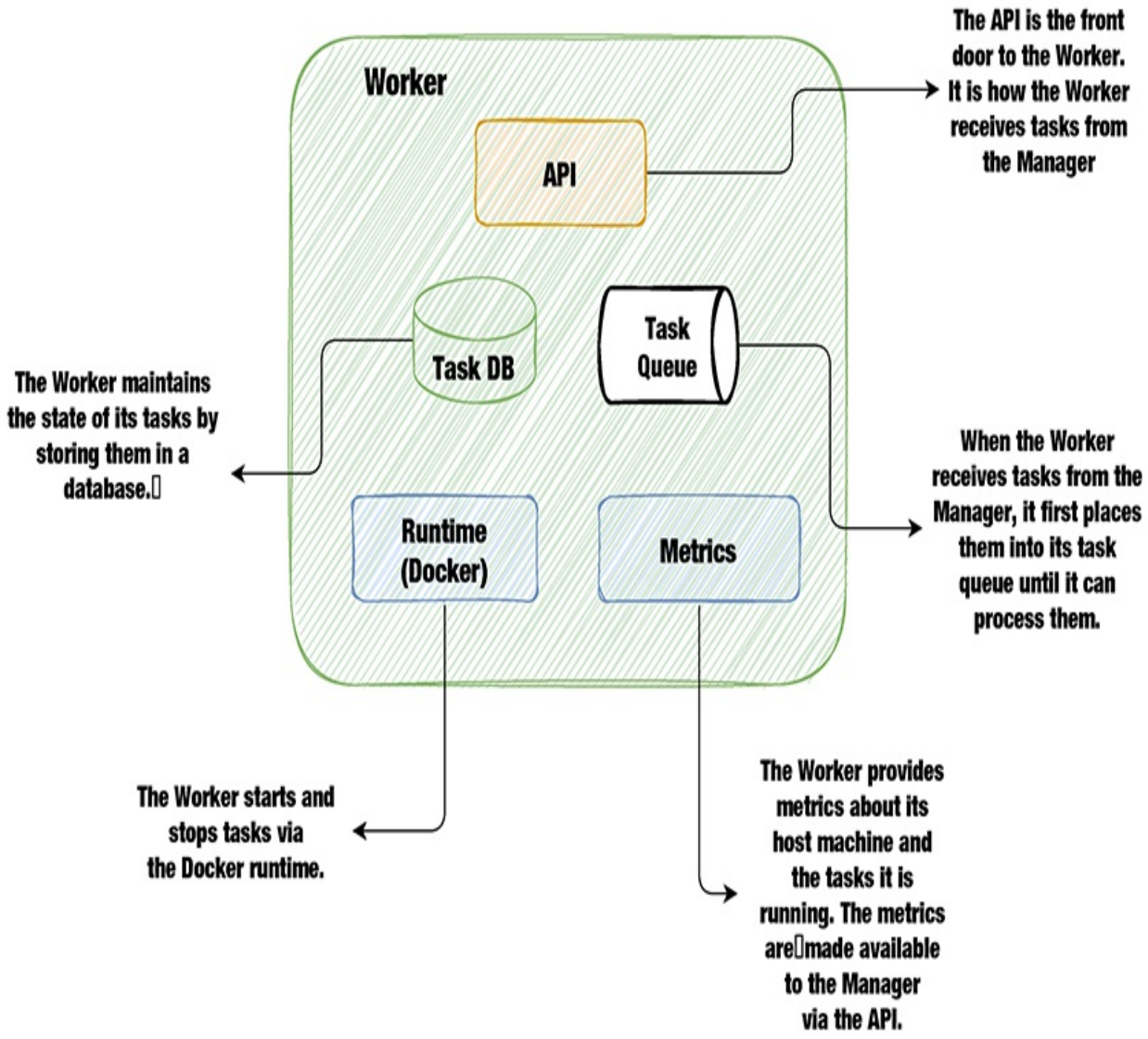


4.1.1 The components that make up the worker

The worker is composed of smaller pieces that perform specific roles. Those pieces, seen in figure 4.3, are an API, a Runtime, a Task Queue, a Task DB,

and Metrics. In this chapter, we're going to focus only on three of these components: the runtime, the task queue, and the task db. We'll work with the other two components in following chapters.

Figure 4.3. Our worker will be made up of these five components, but this chapter will focus only on the *Runtime, Task Queue,* and *Task DB*.



4.2 Tasks and Docker

In chapter 1, we said a *task* is the smallest unit of work in an orchestration system. Then, in chapter 3, we implemented that definition in the `Task` struct,

which we can see again in listing 4.1. This struct is the primary focus of the worker. It receives a task from the manager, then runs it. We'll use this struct throughout this chapter.

As the smallest unit of work, a task performs its work by being run as a Docker container. So, there is a one-to-one correlation between a task and a container.

Listing 4.1. Task struct defined in chapter 2. The worker uses this struct to start and stop tasks.

```
type Task struct {
        ID            uuid.UUID
        ContainerID   string
        Name          string
        State         State
        Image         string
        Memory        int64
        Disk          int64
        ExposedPorts  nat.PortSet
        PortBindings  map[string]string
        RestartPolicy string
        StartTime     time.Time
        FinishTime    time.Time
}
```

In chapter 3, we also defined the `Docker` struct seen in listing 4.2. The worker will use this struct to start and stop the tasks as Docker containers.

Listing 4.2. Docker struct defined in chapter 3.

```
type Docker struct {
        Client      *client.Client
        Config      Config
}
```

The two objects will be the core of the process that will allow our worker to start and stop tasks.

4.3 The role of the queue

Take a peek at listing 4.3 to remind yourself what the `Worker` struct looks

like. The struct is in the same state we left it in chapter 2.

The Worker will use the `Queue` field in the `Worker` struct as a temporary holding area for incoming tasks that need to be processed. When the Manager sends a task to the Worker, the task lands in the queue, which the Worker will process on a first-in-first-out basis.

Listing 4.3. Worker skeleton from chapter 2

```
package worker


import (
    "fmt"

    "github.com/google/uuid"
    "github.com/golang-collections/collections/queue"
)


type Worker struct {
        Name      string
        Queue     queue.Queue
        Db        map[uuid.UUID]*task.Task
        TaskCount int
}
```

It’s important to note that the `Queue` field is itself a struct, which defines several methods we can use to push items onto the queue (`Enqueue`), pop items off of the queue (`Dequeue`), and get the length of the queue (`Len`). The *Queue* field is an example of *composition* in Go. Thus, we can use other structs to compose new, higher level objects.

Also notice that `Queue` is being imported from the *github.com/golang-collections/collections/queue* package. So, we’re re-using a Queue implementation that someone else has written for us. If you haven’t done so already [TODO: reference to an appendix with installation steps], you’ll need to specify this package as a dependency.

4.4 The role of the db

The worker will use the `Db` field to store the state about its tasks. This field is a `map`, where keys are of type `uuid.UUID` from the `github.com/google/uuid` package and values are of type `Task` from our `task` package. There is one thing to note about using a map for the `Db` field. We're starting with a map here out of convenience. This will allow us to write working code quickly. But, this comes with a tradeoff: anytime we restart the worker, we will lose data. This tradeoff is acceptable for the purpose of getting started, but later we'll replace this map with a persistent data store that won't lose data when we restart the worker.

4.5 Counting tasks

Finally, the `TaskCount` field provides a simple count of the tasks the worker has been assigned. We won't make use of this field until the next chapter.

4.6 Implementing the worker's methods

Now that we've reviewed the fields in our `Worker` struct, let's move on and talk about the methods that we stubbed out in chapter 2. The `RunTask`, `StartTask`, and `StopTask` methods seen in listing 4.4 don't do much right now but print out a statement, but by the end of the chapter we will have fully implemented each of them.

Listing 4.4. The stubbed out versions of `RunTask`, `StartTask`, and `StopTask`.

```
func (w *Worker) RunTask() {
    fmt.Println("I will start or stop a task")
}

func (w *Worker) StartTask() {
    fmt.Println("I will start a task")
}

func (w *Worker) StopTask() {
    fmt.Println("I will stop a task")
}
```

We're going to implement these methods in reverse order from what you see above. The reason for implementing them in this order is that the `RunTask`

method will use the other two methods to start and stop tasks.

4.6.1 Implementing the StopTask method

There is nothing complicated about the `StopTask` method. It has a single purpose: to stop running tasks, remembering that a task corresponds to a running container. The implementation, seen in listing 4.5, can be summarized as the following set of steps:

1. Create an instance of the `Docker` struct that allows us to talk to the Docker daemon using the Docker SDK.
2. Call the `Stop()` method on the `Docker` struct.
3. Check if there were any errors in stopping the task.
4. Update the `FinishTime` field on the task `t`.
5. Save the updated task `t` to the worker's `Db` field.
6. Print an informative message and return the result of the operation.

Listing 4.5. Our implementation of the `StopTask` method.

```
func (w *Worker) StopTask(t task.Task) task.DockerResult {
        config := task.NewConfig(&t)
        d := task.NewDocker(config)

        result := d.Stop(t.ContainerID)
        if result.Error != nil {
                log.Printf("Error stopping container %v: %v", t.C
        }
        t.FinishTime = time.Now().UTC()
        t.State = task.Completed
        w.Db[t.ID] = &t
        log.Printf("Stopped and removed container %v for task %v"

        return result
}
```

Notice that the `StopTask` method returns a `task.DockerResult` type. The definition of that type can be seen in listing 4.6. If you remember, Go supports multiple return types We could have enumerated each field in the `DockerResult` struct as a return type to the `StopTask` method. While there is nothing technically wrong with that approach, using the `DockerResult` approach allows us to wrap all the bits related to the outcome of an operation

into a single struct. Anything we want to know about the result of an operation, we simply consult the `DockerResult` struct.

Listing 4.6. A reminder of what the `DockerResult` type looks like.

```
type DockerResult struct {
        Error       error
        Action      string
        ContainerId string
        Result      string
}
```

4.6.2 Implementing the StartTask method

Next, let's implement the `StartTask` method. Similar to the `StopTask` method, `StartTask` is fairly simple, but the process to start a task has a few more steps. The enumerated steps are:

1. Update the `StartTime` field on the task `t`.
2. Create an instance of the `Docker` struct to talk to the Docker daemon.
3. Call the `Run()` method on the `Docker` struct.
4. Check if there were any errors in starting the task.
5. Add the running container's ID to the tasks `t.Runtime.ContainerId` field.
6. Save the updated task `t` to the worker's `Db` field.
7. Return the result of the operation.

The implementation of these steps can be seen in listing 4.7.

Listing 4.7. Our implementation of the `StartTask` method.

```
func (w *Worker) StartTask(t task.Task) task.DockerResult {
        t.StartTime = time.Now().UTC()
        config := task.NewConfig(&t)
        d := task.NewDocker(config)
        result := d.Run()
        if result.Error != nil {
                log.Printf("Err running task %v: %v\n", t.ID, res
                t.State = task.Failed
                w.Db[t.ID] = &t
                return result
```

```
        }

        t.ContainerID = result.ContainerId
        t.State = task.Running
        w.Db[t.ID] = &t

        return result
}
```

By recording the `StartTime` in the `StartTask` method, combined with recording `FinishTime` in the `StopTask` method, we'll later be able to use these timestamps in other output. For example, later in the book we'll write a CLI that allows us to interact with our orchestrator, and the `StartTime` and `FinishTime` values can be output as part of a task's status.

Before we move on from these two methods, I want to point out that neither of them interact directly with the Docker SDK. Instead, they simply call the `Run` and `Stop` methods on the `Docker` object we created. It is the `Docker` object which handles the direct interaction with the Docker client. By encapsulating the interaction with Docker in the `Docker` object, our worker does not need to know anything about the underlying implementation details.

The `StartTask` and `StopTask` methods are the foundation of our Worker. But, in looking at the skeleton we created in chapter two, there is another foundational method missing. How do we add a task to the worker? Remember, we said the worker would use its `Queue` field as a temporary storage for incoming tasks, and when it was ready it would pop a task of the queue and perform the necessary operation.

Let's fix this problem by adding the `AddTask` method seen in listing 4.8. This method performs a single task: it adds the task `t` to the `Queue`.

Listing 4.8. The worker's `AddTask` method.

```
func (w *Worker) AddTask(t task.Task) {
        w.Queue.Enqueue(t)
}
```

4.6.3 An interlude on task state

All that’s left to do now is to implement the `RunTask` method.

Before we do that, however, let’s pause for a moment and recall the purpose of the `RunTask` method. In chapter 2, we said the `RunTask` method "will be responsible for identifying the task’s current state, and then either starting or stopping a task based on the state." But why do we even need `RunTask`?

There are two possible scenarios for handling tasks:

- a task is being submitted for the first time, so the Worker will not know about it
- a task is being submitted for the Nth time, where the task submitted represents the *desired* state to which the *current* task should transition.

When processing the tasks it receives from the Manager, the worker will need to determine which of these scenarios it is dealing with. We’re going to use a naive heuristic to help the worker solve this problem.

Remember that our Worker has the `Queue` and `Db` fields. For our naive implementation, the worker will use the `Queue` field to represent the *desired* state of a task. When the worker pops a task off the queue, it will interpret it as "put task *t* in the state *s*." The worker will interpret tasks it already has in its `Db` field as existing tasks, that is tasks that it has already seen at least once. If a task is in the `Queue` but not the `Db`, then this is the first time the worker is seeing the task, and we default to starting it.

In addition to identifying which of the above two scenarios it is dealing with, the worker will also need to verify if the transition from the *current* state to the *desired* state is a valid one.

Let’s review the states we defined in chapter 2. Listing 4.9 shows that we have states *Pending*, *Scheduled*, *Running*, *Completed*, and *Failed*.

Listing 4.9. The `State` type, which defines the valid states for a task.

```
const (
        Pending State = iota
        Scheduled
        Running
```

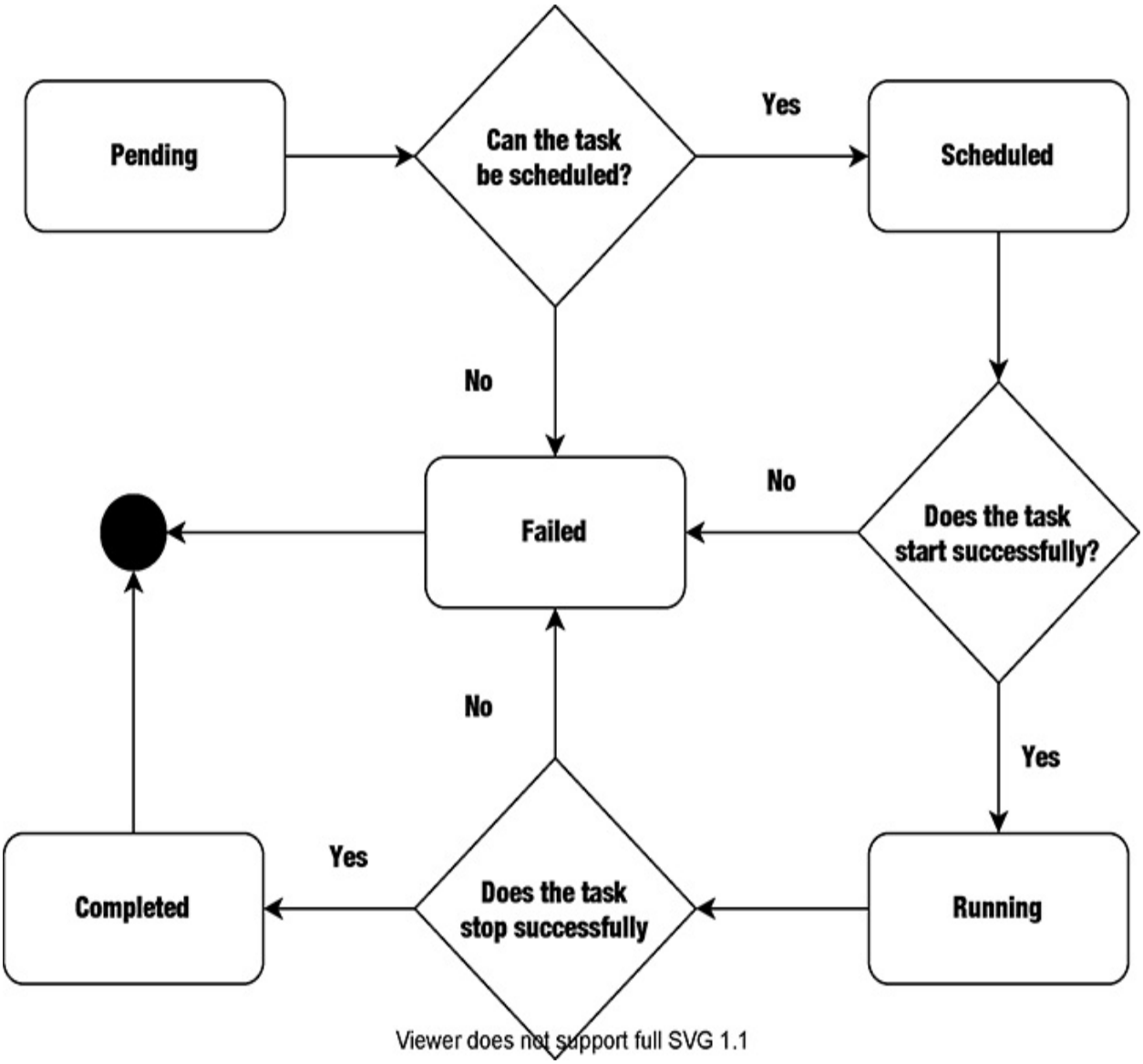
```
        Completed
        Failed
)
```

But what do these states represent? We explained these states in chapter 2, but let’s do a quick refresher:

- *Pending*: this is the initial state, the starting point, for every task.
- *Scheduled*: a task moves to this state once the manager has scheduled it onto a worker.
- *Running*: a task moves to this state when a worker successfully starts the task (i.e. starts the container).
- *Completed*: a task moves to this state when it completes its work in a normal way (i.e. it does not fail).
- *Failed*: if a task does fail, it moves to this state.

To reinforce what these states represent, we can also recall the state diagram from chapter two, seen here in figure 4.4.

Figure 4.4. The states a task will go through during its life cycle.



So, we've defined what the states mean as they relate to a task, but we still haven't defined *how* a task transitions from one state to the next. Nor have we talked about what transitions are valid. For example, if a worker is already running a task—which means it's in the `Running` state—can it transition to the `Scheduled` state? If a task has failed, should it be able to transition from the `Failed` state to the `Scheduled` state?

So before getting back to the `RunTask` method, it looks like we need to figure out this issue of how to handle state transitions. To do this, we can model our states and transitions using the state table seen in table 4.1.

This table has three columns that represent the *CurrentState* of a task, an *Event* that triggers a state transition, and the *NextState* to which the task should transition. Each row in the table represents a specific *valid* transition. Notice that there is not a transition from *Running* to *Scheduled*, or from *Failed* to *Scheduled*.

Table 4.1. State transition table that shows the valid transitions from one state to another.

CurrentState	Event	NextState
Pending	ScheduleEvent	Scheduled
Pending	ScheduleEvent	Failed
Scheduled	StartTask	Running
Scheduled	StartTask	Failed
Running	StopTask	Completed

Now that we have a better understanding of the states and transitions between them, we can translate our understanding into code. Orchestrators like Borg, Kubernetes, and Nomad use a state machine to deal with the issue of state transitions. However, in order to keep the number of concepts and technologies we have to deal with to a minimum, we're going to hard code our state transitions into the `stateTransitionMap` type you see in listing 4.10. This map encodes the transitions we identified above in table 4.1.

The `stateTransitionMap` creates a map between a `State` and a slice of states, `[]State`. Thus, the keys in this map are the current state, and the values are the valid transition states. For example, the *Pending* state can only transition to the *Scheduled* state. The *Scheduled* state, however, can transition

to *Running*, *Completed*, or *Failed*.

Listing 4.10. The stateTransitionMap map.

```
var stateTransitionMap = map[State][]State{
        Pending:   []State{Scheduled},
        Scheduled: []State{Scheduled, Running, Failed},
        Running:   []State{Running, Completed, Failed},
        Completed: []State{},
        Failed:    []State{},
}
```

In addition to `stateTransitionMap`, we're going to implement the `Contains` and `ValidStateTransition` helper functions seen in listing 4.11. These functions will perform the actual logic to verify that a task can transition from one state to the next.

Let's start with the `Contains` function. It takes two arguments: *states*, a slice of type `State`, and *state* of type `State`. If it finds *state* in the slice of *states*, it returns `true`, otherwise it returns false.

The `ValidStateTransition` function is a wrapper around the *Contains* function. It provides a convenient way for callers of the function to simply ask, "Hey, can a task transition from this state to that state?" All the heavy lifting is done by the *Contains* function.

You should add the code is listing 4.8 to the `state.go` file in the `task` directory of your project.

Listing 4.11. Helper methods

```
func Contains(states []State, state State) bool {
        for _, s := range states {
                if s == state {
                        return true
                }
        }
        return false
}

func ValidStateTransition(src State, dst State) bool {
        return Contains(stateTransitionMap[src], dst)
```

```
}
```

4.6.4 Implementing the RunTask method

Now we can finally talk more specifically about the *RunTask* method. It took us a while to get here, but we needed to iron out those other details before it even made sense talking about this method. And because we did that leg work, implementing the *RunTask* method will go a bit more smoothly.

As we said earlier in the chapter, the `RunTask` method will identify the task's current state and then either start or stop it based on that state. We can use a fairly naive algorithm to determine whether the worker should start or stop a task. It looks like this:

1. Pull a task of the queue.
2. Convert it from an interface to a task.Task type.
3. Retrieve the task from the worker's `Db`.
4. Check if the state transition is valid.
5. If the task from the queue is in a state `Scheduled`, call `StartTask`.
6. If the task from the queue is in a state `Completed`, call `StopTask`.
7. Else there is an invalid transition, so return an error.

All that's left to do now is to implement the above steps in our code, which can be seen in figure 4.12.

Listing 4.12. Our implementation of the `RunTask` method.

```
func (w *Worker) RunTask() task.DockerResult {
        t := w.Queue.Dequeue() #1
        if t == nil {
                log.Println("No tasks in the queue")
                return task.DockerResult{Error: nil}
        }

        taskQueued := t.(task.Task) #2

        taskPersisted := w.Db[taskQueued.ID] #3
        if taskPersisted == nil {
                taskPersisted = &taskQueued
                w.Db[taskQueued.ID] = &taskQueued
        }
```

```
        var result task.DockerResult
        if task.ValidStateTransition(taskPersisted.State, taskQue
                switch taskQueued.State {
                case task.Scheduled:
                        result = w.StartTask(taskQueued) #5
                case task.Completed:
                        result = w.StopTask(taskQueued) #6
                default:
                        result.Error = errors.New("We should not
                }
        } else {
                err := fmt.Errorf("Invalid transition from %v to
                result.Error = err #7
        }
        return result #8
}
```

4.7 Putting it all together

Whew, we've made it. We covered a lot of territory in implementing the methods for our worker. If you remember chapter 3, we ended by writing a program that used the work we did earlier in the chapter. We're going to continue that practice in this chapter.

Before we do, however, remember that in chapter 3 we built out the `Task` and `Docker` structs, and that work allowed us to start and stop containers. The work we did in this chapter sits on top of the work from the last chapter. So once again, we're going to write a program that will start and stop…*Tasks*. The worker operates on the level of the `Task`, and the `Docker` struct operates on the lower level of the container.

Now, let's write a program to pull everything together into a functioning worker. You can either comment out the code from the `main.go` file you used in the last chapter, or create a new `main.go` file to use for this chapter.

The program is simple. We create a worker `w`, which has a `Queue` and `Db` fields like we talked about at the beginning of the chapter. Next, we create a task `t`. This task starts with a state of `Scheduled`, and it uses a Docker image named `strm/helloworld-http`. More about this image in a bit. After creating a worker and a task, we call the worker's `AddTask` method and pass it the task

`t`. Then it calls the worker's `RunTask` method. This method will pull the task `t` off the queue and do the right thing. It captures the return value from the `RunTask` method and stores it in the variable `result`. (Bonus points if you remember what type is returned from `RunTask`.)

At this point, we have a running container.

After sleeping for 30 seconds (feel free to change the sleep time to whatever you want), then we start the process of stopping the task. We change the task's state to `Completed`, call `AddTask` again and pass it the same task, and finally call `RunTask` again. This time when `RunTask` pulls the task off the queue, the task will have a container ID and a different state. As a result, the task gets stopped.

Listing 4.13 shows our program to create a worker, add a task, start it, and finally stop it.

Listing 4.13. This program pulls everything together into a functioning worker that starts and stops a task.

```
// previous code not shown

func main() {
        db := make(map[uuid.UUID]*task.Task)
        w := worker.Worker{
                Queue: *queue.New(),
                Db:    db,
        }

        t := task.Task{
                ID:    uuid.New(),
                Name:  "test-container-1",
                State: task.Scheduled,
                Image: "strm/helloworld-http",
        }

        // first time the worker will see the task
        fmt.Println("starting task")
        w.AddTask(t)
        result := w.RunTask()
        if result.Error != nil {
                panic(result.Error)
        }
```

```
        t.ContainerID = result.ContainerId

        fmt.Printf("task %s is running in container %s\n", t.ID,
        fmt.Println("Sleepy time")
        time.Sleep(time.Second * 30)

        fmt.Printf("stopping task %s\n", t.ID)
        t.State = task.Completed
        w.AddTask(t)
        result = w.RunTask()
        if result.Error != nil {
                panic(result.Error)
        }
}
```

Let's pause for a moment and talk about the image used in the above code listing. At the beginning of the chapter, we talked about the scenario of scaling a static website using an orchestrator, specifically the worker component. This image, `strm/helloworld-http` provides a concrete example of a static website: it runs a web server that serves a single file. To verify this behavior, when you run the program, in a separate terminal type the `docker ps` command. You should see output similar to listing 4.14. In that output, you can find the port the web server is running on by looking at the `PORTS` column. Then, open your browser and type `localhost:<port>`. In the case of the output below, I would type `localhost:49161` in my browser.

Listing 4.14. Output from the `docker ps` command. The output has been truncated to make it more readable.

```
$ docker ps
CONTAINER ID   IMAGE                  PORTS                    NAMES
4723a4201829   strm/helloworld-http   0.0.0.0:49161->80/tcp    test-
```

When I browse to the server on my machine, I see "Hello from 90566e236f88".

Go ahead and run the program. You should see output similar to that in listing 4.15.

Listing 4.15. Running your main program should start the task, sleep for a bit, then stop the task.

```
$ go run main.go
starting task
{"status":"Pulling from strm/helloworld-http","id":"latest"}
{"status":"Digest: sha256:bd44b0ca80c26b5eba984bf498a9c3bab0eb1c5
{"status":"Status: Image is up to date for strm/helloworld-http:l
task bfe7d381-e56b-4c4d-acaf-cbf47353a30a is running in container
Sleepy time
stopping task bfe7d381-e56b-4c4d-acaf-cbf47353a30a
2021/08/08 21:13:09 Attempting to stop container e13af1f4b9cbac6f
2021/08/08 21:13:19 Stopped and removed container e13af1f4b9cbac6
```

Congratulations! You now have a functional worker. Before moving on to the next chapter, play around with what you've built. In particular, modify the `main` function from listing 4.13 to create multiple workers, then add tasks to each them.

4.8 Summary

- Tasks are executed as containers, meaning there is a one-to-one relationship between a task and a container.
- The worker performs two basic actions on tasks, either starting or stopping them. These actions result in tasks transitioning from one state to the next valid state.
- The worker shows how the Go language supports object composition. The worker itself is a composition of other objects, in particular the Worker's `Queue` field is a struct defined in the *github.com/golang-collections/collections/queue* package
- The worker, as we've designed and implemented it, is simple. We've used clear and concise processes that are easy to implement in code.
- The worker does not interact directly with the Docker SDK. Instead, it uses our `Docker` struct, which is a wrapper around the SDK. By encapsulating the interaction with the SDK in the `Docker` struct, we can keep the `StartTask` and `StopTask` methods small and readable.

5 An API for the worker

This chapter covers

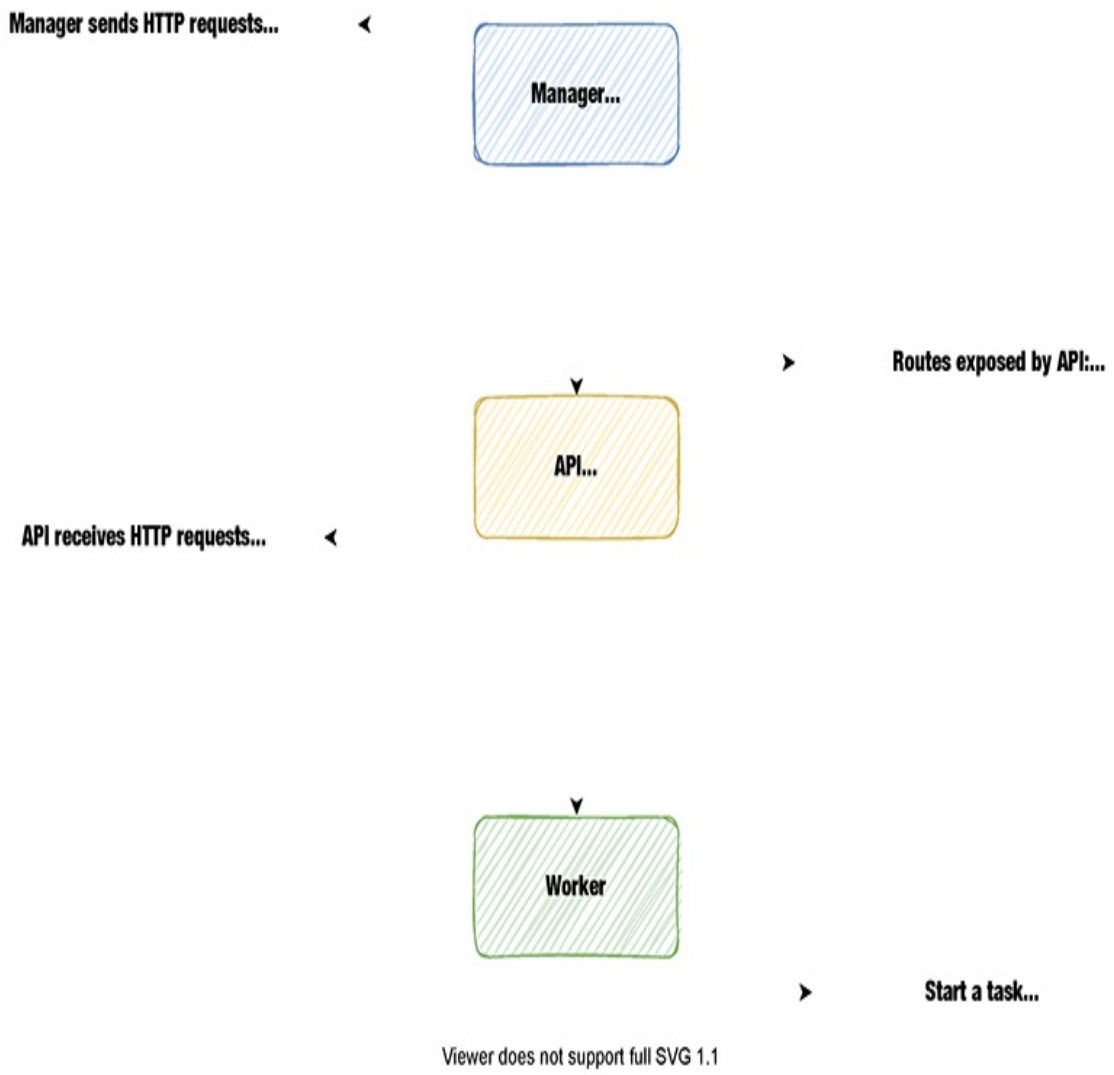
- Understanding the purpose of the worker API
- Implementing methods to handle API requests
- Creating a server to listen for API requests
- Starting, stopping, and listing tasks via the API

In chapter 4, we implemented the core features of the worker: pulling tasks off its queue, and then starting or stopping them. Those core features alone, however, do not make the worker complete. We need a way to expose those core features so a manager, which will be the exclusive user, running on a different machine can make use of them. To do this, we're going to wrap the worker's core functionality in an Application Programming Interface, or API.

The API will be simple, as you can see if figure 5.1, providing the means for a manager to perform these basic operations:

- Send a task to the worker (which results in the working starting the task as a container)
- Get a list of the worker's tasks
- Stop a running task

Figure 5.1. The API for our orchestrator provides a simple interface to the worker.



5.1 Overview of the worker API

We've enumerated above the operations that the worker's API will support: sending a task to a worker to be started, getting a list of tasks, and stopping a task. But, how will we implement those operations? We're going to implement those operations using a web API. This choice means that the worker's API can be exposed across a network, and that it will use the HTTP protocol. Like most web APIs, the worker's API will use three primary components:

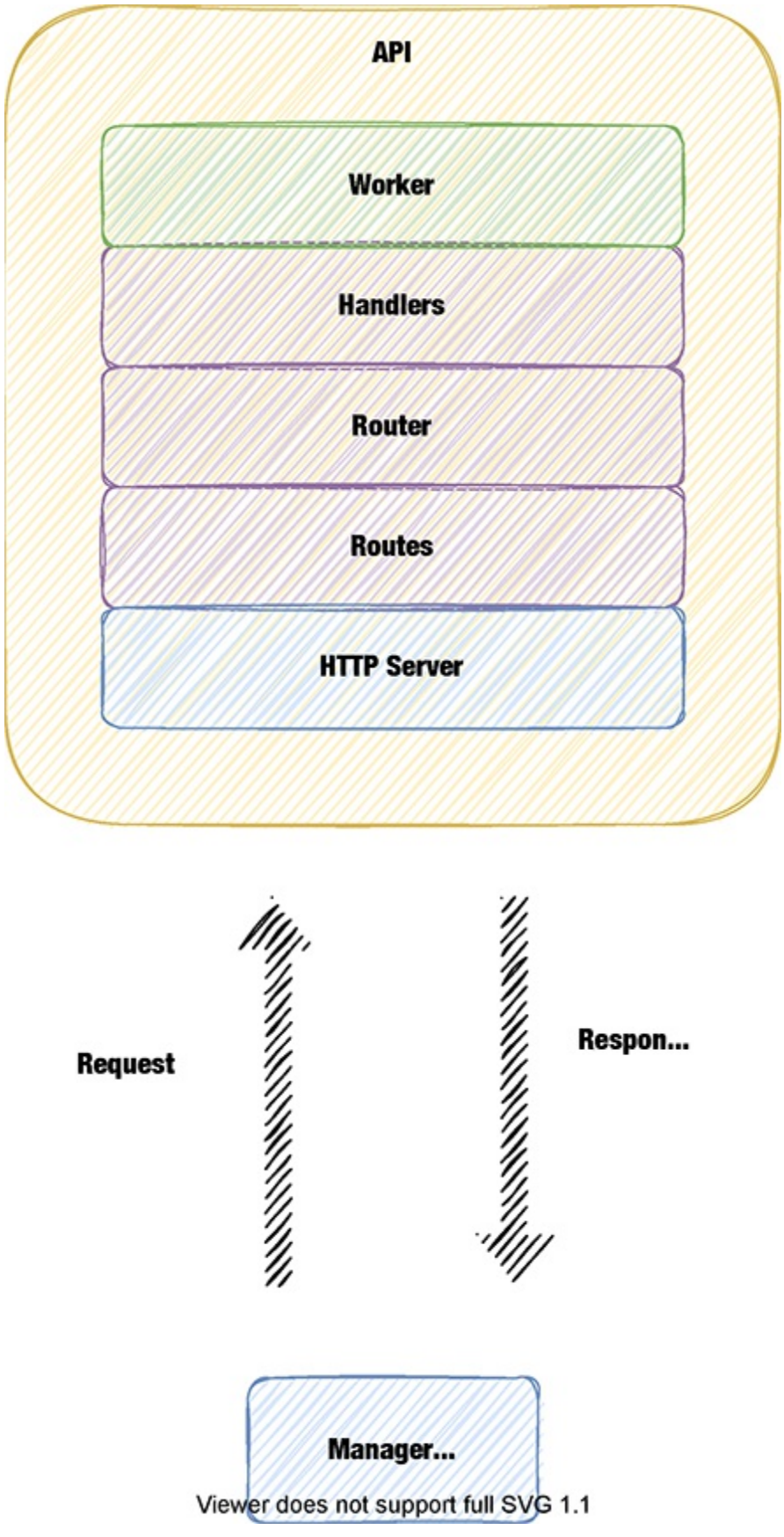
- *Handlers*: functions that are capable of responding to requests.
- *Routes*: patterns that can be used to match the URL of incoming

requests.
- *Router*: an object that uses routes to match incoming requests with the appropriate handler

We could implement our API using nothing but the http package from the Go standard library. That can be tedious, however, because the http package is missing one critical piece: the ability to define *parameterized routes*. What is a parameterized route? It's a route that defines a URL where one or more parts of the URL path are unknown and may change from one request to the next. This is particular useful for things like identifiers. For example, if a route like `/tasks` called with a HTTP `GET` request returns a list of all tasks, then a route like `/tasks/5` returns a single item, the task whose identifier is the integer five. Since each task should have a unique identifier, however, we need to provide a pattern when defining this kind of route in a web API. The way to do this is to use a parameter for the part of the URL path that can be different with each request. In the case of tasks, we can use a route defined as `/tasks/{taskID}`.

Because the http package from the standard library doesn't provide a robust and easy way to define parameterized routes, we're going to use a lightweight, third-party router called chi. Conceptually, our API will look like what you see in figure 5.2. Requests will be sent to an HTTP server, which you can see in the blue box. This server provides a function called *ListenAndServe* which is the lowest layer in our stack and will handle the low level details of listening for incoming requests. The next three layers—Routes, Router, and Handlers—are all provided by chi.

Figure 5.2. Internally, the worker's API is composed of an HTTP server from the Go standard library; the routes, router, and handlers from the chi package; and finally our own worker.


API
Worker
Handlers
Router
Routes
HTTP Server
Request
Respon...
Manager...
Viewer does not support full SVG 1.1

For our worker API we'll use the routes defined below in table 5.1. Because we're exposing our worker's functionality via a web API, the routes will involve standard HTTP methods like GET, POST, and DELETE. The first route in the table, `/tasks`, will use the HTTP `GET` method and will return a list of tasks. The second route is the same as the first, but it uses the `POST` method, which will start a task. The third route, `/tasks/{taskID}`, will stop the running task identified by the parameter `{taskID}`.

Table 5.1. Routes used by our worker API

Method	Route	Description
GET	/tasks	get a list of all tasks
POST	/tasks	create a task
DELETE	/tasks/{taskID}	stop the task identified by `taskID`

If you work with REST (Representational State Transfer) APIs at your day job, this should look familiar. If you're not familiar with REST, or you are new to the world of APIs, don't worry. It's not necessary to be a REST expert to grok what we're building in this chapter. At a very hand-wavy level, REST is an architectural style that builds on the client-server model of application development. If you want to learn more about REST, you can start with a gentle introduction like the API 101: What Is a REST API? post on the Postman blog.

5.2 Data format, requests, and responses

Before we get into writing any code, we need to address one more important item. From your experience browsing the internet, you know that when you

type an address into your browser, you get back data. Type https://espn.com and you get data about sports. Type https://nytimes.com and you get data about current events. Type in https://www.funnycatpix.com and you get data that is pictures of cats.

Like the above websites, the worker API deals with data, both sending and receiving data. It's data, however, is not about news or cats but about tasks. Furthermore, the data the worker API deals with will take a specific form, and that form is JSON, which stands for *Javascript Object Notation*. You're probably already familiar with JSON, as it's the lingua franca of many modern APIs. This decision has two consequences:

1. any data sent to the API—e.g for our `POST /tasks` route in the table above—must be encoded as JSON data in the body of the request; and
2. any data returned from the API—e.g. for our `GET /tasks` route—must be encoded as JSON data in the body of the response.

For our worker, we only have one route, the `POST /tasks` route, that will accept a body. But what data does our worker expect to be in the body of that request?

If you remember from the last chapter, the Worker has a `StartTask` method that takes a `task.Task` type as an argument. That type holds all the necessary data we need to start the task as a Docker container. But what the Worker API will receive (from the Manager) is a task.TaskEvent type, which contains a `task.Task`. So, the job of the API is to extract that task from the request and add it to the worker's queue. Thus, a request to our `POST /tasks` route will look like that in listing 5.1.

Listing 5.1. The Worker API receives a task.TaskEvent from the Manager. The task.TaskEvent here was used in chapter 4.

```
{
    "ID": "6be4cb6b-61d1-40cb-bc7b-9cacefefa60c",
    "State": 2,
    "Task": {
        "State": 1,
        "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
        "Name": "test-chapter-5",
        "Image": "strm/helloworld-http"
```

```
    }
}
```

The response to our `POST /tasks` request will have a status code of `201` and includes a JSON-encoded representation of the task in the response body. Why a `201` and not a `200` response code? We could use a `200` response status. According to the HTTP spec described in RFC 7231, "The 200 (OK) status code indicates that the request has succeeded. The payload sent in a 200 response depends on the request method". (https://datatracker.ietf.org/doc/html/rfc7231#section-6.3.1) Thus, the `200` response code is the generic case telling the requester, "Yes, I received your request and it was successful". The `201` response code, however, handles the more specific case, i.e. for a POST request, that tells the requester, "Yes, I received your request and I created a new resource". In our case, that new resource is the task sent in the request body.

Like the `POST /tasks` route, the `GET /tasks` route returns a body in its response. This route ultimately calls the `GetTasks` method on our worker, which returns a slice of pointers to `task.Task` types, effectively a list of tasks. Our API in this situation will take that slice returned from `GetTasks`, encode it as JSON and then return it. Listing 5.2 shows an example of what such a response might look.

Listing 5.2. The Worker API returns a list of tasks for the GET /tasks route. In this example, there are two tasks.

```
[
  {
    "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
    "ContainerID": "4f67af51b173564ffd50a3c7fdec258321262f5fa0529
    "Name": "test-chapter-5",
    "State": 2,
    "Image": "strm/helloworld-http",
    "Memory": 0,
    "Disk": 0,
    "ExposedPorts": null,
    "PortBindings": null,
    "RestartPolicy": "",
    "StartTime": "0001-01-01T00:00:00Z",
    "FinishTime": "0001-01-01T00:00:00Z"
  },
  {
```

```
    "ID": "266592cd-960d-4091-981c-8c25c44b1018",
    "ContainerID": "180d207fa788d5261e6ccf927012476e24286c07fc3a3
    "Name": "test-chapter-5-1",
    "State": 2,
    "Image": "strm/helloworld-http",
    "Memory": 0,
    "Disk": 0,
    "ExposedPorts": null,
    "PortBindings": null,
    "RestartPolicy": "",
    "StartTime": "0001-01-01T00:00:00Z",
    "FinishTime": "0001-01-01T00:00:00Z"
  }
]
```

In addition to the list of tasks, the response will also have a status code of `200`.

Finally, let's talk about the `DELETE /tasks/{taskID}` route. Like the `GET /tasks` route, this one will not take a body in the request. Remember, we said earlier that the `{taskID}` part of the route is a parameter and allows the route to be called with arbitrary IDs. So, this route allows us to stop a task for the given `taskID`. This route will only return a status code of `204`; it will not include a body in the response.

So, with this new information, let's update table 5.1.

Table 5.2. Updated table 5.1 that for each request additionally shows whether the route accepts a request body, whether it returns a response body, and what status code is returned for a successful request.

Method	Route	Description	Request Body	Response Body	Status code
GET	/tasks	get a list of all tasks	none	list of tasks	200
POST	/tasks	create a task	JSON-encoded task.TaskEvent	none	201

DELETE	/tasks/{taskID}	stop the task identified by `taskID`	none	none	204

5.3 The API struct

At this point, we've set the stage for writing the code for the worker API. We've identified the main components of our API, defined the data format used by that API, and enumerated the routes the API will support.

We're going to start by representing our API in code as the struct seen in listing 5.3. You should create a file named `api.go` in the `worker/` directory of your code where you can place this struct.

This struct serves several purposes. First, it contains the `Address` and `Port` fields, which define the local IP address of the machine where the API runs and the port on which the API will listen for requests. These fields will be used to start the API server, which we will implement later in the chapter. Second, it contains the `Worker` field, which will be a reference to an instance of a Worker object. Remember, we said the API will wrap the worker in order to expose the Worker's core functionality to the Manager. This field is the means by which that functionality is exposed. Third, the struct contains the `Router` field, which is a pointer to an instance of `chi.Mux`. This field is what brings in all the functionality provided by the chi router.

Listing 5.3. The API struct that will power our worker.

```
type Api struct {
        Address string
        Port    int
        Worker  *Worker
        Router  *chi.Mux
}
```

What is a mux?

The term *mux* stands for multiplexer and can be used synonymously with *request router*.

5.4 Handling Requests

With the API struct defined, we've given the API a general shape, or form, at a high level. This shape is what will contain the API's three components: handlers, routes, and a router. Let's dive deeper into the API and implement the handlers that will be able to respond to the routes we defined in table 5.1 above.

As we've already said, a *handler* is a function capable of responding to a request. In order for our API to handle incoming requests, we need to define handler methods on the `API` struct. We're going to use the following three methods, which I'll list here with their method signatures:

- StartTaskHandler(w http.ResponseWriter, r *http.Request)
- GetTasksHandler(w http.ResponseWriter, r *http.Request)
- StopTaskHandler(w http.ResponseWriter, r *http.Request)

There is nothing terribly complicated about these handler methods. Each method takes the same arguments, an *http.ReponseWriter* type and a pointer to an *http.Request* type. Both of these types are defined in the *http* package in Go's standard library. The http.ResponseWriter `w` will contain data related to responses. The http.Request `r` will hold data related to requests.

To implement these handlers, create a file named `handlers.go` in the `worker` directory of your project, then open that file in a text editor. We'll start by adding the *StartTaskHandler* method seen in listing 5.4. At a high level, this method reads the body of a request from `r.Body`, converts the incoming data it finds in that body from JSON to an instance of our `task.TaskEvent` type, then adds that `task.TaskEvent` to the worker's queue. It wraps up by printing a log message and then adding a response code to the `http.ResponseWriter`.

Listing 5.4. The worker's StartTaskHandler method takes incoming requests to start a task, reads the body of the request and converts it from JSON to a task.TaskEvent, then puts that on the worker's queue.

```
func (a *Api) StartTaskHandler(w http.ResponseWriter, r *http.Req
        d := json.NewDecoder(r.Body) #1
        d.DisallowUnknownFields() #2

        te := task.TaskEvent{} #3
        err := d.Decode(&te) #4
        if err != nil { #5
                msg := fmt.Sprintf("Error unmarshalling body: %v\
                log.Printf(msg)
                w.WriteHeader(400)
                e := ErrResponse{
                        HTTPStatusCode: 400,
                        Message:        msg,
                }
                json.NewEncoder(w).Encode(e)
                return
        }

        a.Worker.AddTask(te.Task) #6
        log.Printf("Added task %v\n", te.Task.ID) #7
        w.WriteHeader(201) #8
        json.NewEncoder(w).Encode(te.Task) #9
}
```

The next method we'll implement is the `GetTasksHandler` method in listing 5.5. This method looks simple, but there is a lot going inside it. It starts off by setting the `Content-Type` header to let the client know we're sending it JSON data. Then, similar to `StartTaskHandler`, it adds a response code. And then we come to the final line in the method. It may look a little complicated, but it's really just a compact way to express the following operations:

- get an instance of a `json.Encoder` type by calling the `json.NewEncoder()` method
- get all the worker's tasks by calling the worker's `GetTasks` method
- transform the list of tasks into JSON by calling the `Encode` method on the `json.Encoder` object

Listing 5.5. The worker's GetTasksHandler does what it advertises: it returns all of the tasks the worker knows about.

```
func (a *Api) GetTasksHandler(w http.ResponseWriter, r *http.Requ
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(200)
        json.NewEncoder(w).Encode(a.worker.GetTasks())
```

```
}
```

The final handler to implement is the `StopTaskHandler`. If we glance back at table 5.2, we can see stopping a task is accomplished by sending a request with a path of `/tasks/{taskID}`. An example of what this path will look like when a real request is made is `/tasks/6be4cb6b-61d1-40cb-bc7b-9cacefefa60c`. This is all that's needed to stop a task, because the worker already knows about the task: it has it stored in its `Db` field.

The first thing the `StopTaskHandler` must do is read the `taskID` from the request path. As you can see in figure 5.6, we're doing that by using a helper function named `URLParam` from the chi package. We're not going to worry about how the helper method is getting the taskID for us; all we care about is that it simplifies our life a bit and gives us the data we need to get on with the job of stopping a task.

Now that we have the taskID, we have to convert it from a string, which is the type that `chi.URLParam` returns to us, into a uuid.UUID type. This conversion is done by calling the `uuid.Parse()` method, passing it the string version of the taskID. Why do we have to perform this step? It's necessary because the worker's `Db` field is a map which has keys of type `uuid.UUID`. So if we were to try to lookup a task using a string, the compiler would yell at us.

Okay, so now we have a taskID and we have converted it to the correct type. The next thing we want to do is to check if the worker actually knows about this task. If it doesn't, then we should return a response with a `404` status code. If it does, then we change the state to `task.Completed` and add it to the worker's queue. This is what the remaining of the method is doing.

Listing 5.6. The worker's StopTaskHandler uses the taskID from the request path to add a task to the worker's queue that will stop the specified task.

```
func (a *Api) StopTaskHandler(w http.ResponseWriter, r *http.Requ
        taskID := chi.URLParam(r, "taskID")  #1
        if taskID == "" {  #2
                log.Printf("No taskID passed in request.\n")
                w.WriteHeader(400)
        }
```

```
        tID, _ := uuid.Parse(taskID)  #3
        _, ok := a.Worker.Db[tID]  #4
        if !ok {  #5
                log.Printf("No task with ID %v found", tID)
                w.WriteHeader(404)
        }

        taskToStop := a.Worker.Db[tID]  #6
        taskCopy := *taskToStop  #7
        taskCopy.State = task.Completed  #8
        a.Worker.AddTask(taskCopy)  #9

        log.Printf("Added task %v to stop container %v\n", taskTo
        w.WriteHeader(204)  #11
}
```

There is one little gotcha in our `StopTaskHandler` that's worth explaining in more detail. Notice that we're making a copy of the task that's in the worker's datastore. Why is this necessary?

As we mentioned in chapter 4, we're using the worker's datastore to represent the current state of tasks, while we're using the worker's queue to represent the desired state of tasks. As a result of this decision, the API cannot simply retrieve the task from the worker's datastore, set the state to `task.Completed`, then put the task onto the worker's queue. The reason for this is that the values in datastore are pointers to `task.Task` types. If we were to change the state on `taskToStop`, we would be changing the state field on the task in the datastore. We would then add the same task to the worker's queue, and when it popped the task off to work on it, it would complain about not being able to transition a task from the state `task.Completed` to `task.Completed`. Hence, we make a copy, change the state on the copy, and add it to the queue.

5.5 Serving the API

Up to this point, we've been setting the stage for serving the worker's API. We've created our `API` struct that contains the two components that will make this possible: that is, the `Worker` and `Router` fields. Each of these is a pointer to another type. The `Worker` field is a pointer to our own `Worker` type that we created in chapter 3, and it will provide all the functionality to start and stop

tasks and get a list of tasks the worker knows about. The `Router` field is a pointer to a `Mux` object provided by the `chi` package, and it will provide the functionality for defining routes and routing requests to the handlers we defined earlier.

In order to serve the worker's API, we need to make two additions to the code we've written so far. Both additions will be made to the `api.go` file.

The first addition is to add the `initRouter()` method to the `Api` struct as you see in listing 5.7. This method, as its name suggests, initializes our router. It starts by creating an instance of a `Router` by calling `chi.NewRouter()`. Then it goes about setting up the routes we defined above in table 5.2. We won't get into the internals of how the chi package creates these routes.

Listing 5.7. The initRouter() method.

```
func (a *Api) initRouter() {
        a.Router = chi.NewRouter() #1
        a.Router.Route("/tasks", func(r chi.Router) { #2
                r.Post("/", a.StartTaskHandler) #3
                r.Get("/", a.GetTasksHandler)
                r.Route("/{taskID}", func(r chi.Router) { #4
                        r.Delete("/", a.StopTaskHandler) #5
                })
        })
}
```

The final addition is to add the `Start()` method to the `Api` struct as you see in listing 5.8. This method calls the `initRouter` method defined in listing 5.4, and then it starts an HTTP server that will listen for requests. The `ListenAndServe` function is provided by the `http` package from Go's standard library. It takes an address, which we're building with the `fmt.Sprintf` function and will typically (for us) look like `127.0.0.1:5555`, and a handler, which for our purposes is the router that gets created in the `initRouter()` method.

Listing 5.8. The Start() method initializes our router and starts listening for requests

```
func (a *Api) Start() {
        a.initRouter()
```

```
        http.ListenAndServe(fmt.Sprintf("%s:%d", a.Address, a.Por
}
```

5.6 Putting it all together

Like we've done in previous chapters, it's time to take the code we've written and actually run it. To do this, we're going to continue our use of `main.go`, in which we'll write our `main` function. You can either re-use the `main.go` file from the last chapter and just delete the contents of the main function, or start with a fresh file.

In your `main.go` file, add the `main()` function from listing 5.9. This function uses all the work we've done up to this point. It creates an instance of our Worker, `w`, which has a `Queue` and a `Db`. It creates an instance of our Api, `api`, which uses the `host` and `port` values that it reads from the local environment.

Finally, the `main()` function performs the two operations that brings everything to life.

The first of these operations is to call a function `runTasks` and passing it a pointer to the worker `w`. But it also does something else. It has this funny `go` term before calling the `runTasks` function. What is that about? If you've used threads in other languages, the `go runTasks(&w)` line is similar to using threads. In Go, threads are called *goroutines*, and they provide the ability to perform concurrent programming. We won't go into the details of goroutines here, because there are other resources dedicated solely to this topic. For our purposes, all we need to know is that we're creating a goroutine and inside it we will run the `runTasks` function. After creating the goroutine, we can continue on in the main function and start our API by calling `api.Start()`.

Listing 5.9. Running our worker from main.go.

```
func main() {
        host := os.Getenv("CUBE_HOST")
        port, _ := strconv.Atoi(os.Getenv("CUBE_PORT"))

        fmt.Println("Starting Cube worker")

        w := worker.Worker{
```

```
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]*task.Task),
        }
        api := worker.Api{Address: host, Port: port, Worker: &w}

        go runTasks(&w)
        api.Start()
}
```

Now, let's talk about the `runTasks` function, which you can see in listing 5.10. This function runs in a separate goroutine from the main function, and it's fairly simple. It's a continuous loop that checks the worker's queue for tasks and calls the worker's `RunTask` method when it finds tasks that need to run. For our own convenience, we're sleeping for ten seconds between each iteration of the loop. This slows things down for us so we can easily read any log messages.

Listing 5.10. The runTasks function.

```
func runTasks(w *worker.Worker) {
        for {
                if w.Queue.Len() != 0 {
                        result := w.RunTask()
                        if result.Error != nil {
                                log.Printf("Error running task: %
                        }
                } else {
                        log.Printf("No tasks to process currently
                }
                log.Println("Sleeping for 10 seconds.")
                time.Sleep(10 * time.Second)
        }

}
```

There is a reason that we've structured our main function like this. If you recall the handler functions we wrote earlier in the chapter, they were performing a very narrow set of operations, namely:

- reading requests sent to the server
- getting a list of tasks from the worker (in the case of the `GetTasksHandler`)
- putting a task on the worker's queue

- sending a response to the requester

Notice that the API is not calling any worker methods that perform task operations, that is it is not starting or stopping tasks. Structuring our code in this way allows us to separate the concern of handling requests from the concern of performing the operations to start and stop tasks. Thus, we make it easier on ourselves to reason about our codebase. If we want to add a feature or fix a bug with the API, we know we need to work in the `api.go` file. If we want to do the same for request handling, we need to work in the `handlers.go` file. And, for anything related to the operations of starting and stopping tasks, we need to work in the `worker.go` file.

Okay, time to make some magic. Running our code should result in a number of log messages being printed to the terminal, like this:

```
$ go run main.go #1
Starting Cube worker #2
2021/11/05 14:17:53 No tasks to process currently. #3
2021/11/05 14:17:53 Sleeping for 10 seconds.
2021/11/05 14:18:03 No tasks to process currently. #4
2021/11/05 14:18:03 Sleeping for 10 seconds.
2021/11/05 14:18:13 No tasks to process currently. #5
2021/11/05 14:18:13 Sleeping for 10 seconds.
```

As you can see when we first start the worker API, it doesn't do much. It tells us it doesn't have any tasks to process, then sleeps for ten seconds, and then wakes up again and tells us the same thing. This isn't very exciting. Let's spice things up by interacting with the worker API. We'll start with getting a list of tasks using the `curl` command in a separate terminal:

```
$ curl -v localhost:5555/tasks #1
*   Trying 127.0.0.1:5555...
* Connected to localhost (127.0.0.1) port 5555 (#0) #2
> GET /tasks HTTP/1.1 #3
> Host: localhost:5555
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK #4
< Content-Type: application/json
< Date: Fri, 05 Nov 2021 18:27:25 GMT
```

```
< Content-Length: 3
<
[] #5
```

Great! We can query the API to get a list of tasks. As expected, though, the response is an empty list because the worker doesn't have any tasks yet. Let's remedy that by sending it a request to start a task.

```
curl -v --request POST \  #1
  --header 'Content-Type: application/json' \  #2
  --data '{  #3
    "ID": "266592cd-960d-4091-981c-8c25c44b1018",
    "State": 2,
    "Task": {
        "State": 1,
        "ID": "266592cd-960d-4091-981c-8c25c44b1018",
        "Name": "test-chapter-5-1",
        "Image": "strm/helloworld-http"
    }
}
' localhost:7777/tasks
```

When you run the above `curl` command, you should see output like that below. Notice that the status code in the response is `HTTP/1.1 201 Created` and there is no response body.

```
*   Trying 127.0.0.1:5555...
* Connected to localhost (127.0.0.1) port 5555 (#0)
> POST /tasks HTTP/1.1
> Host: localhost:5555
> User-Agent: curl/7.80.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 243
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 201 Created
< Date: Mon, 29 Nov 2021 22:24:51 GMT
<
* Connection #0 to host localhost left intact
```

At the same time you run the `curl` command above, you should see log messages in the terminal where the API is running. Those log messages should look like this:

```
2021/11/05 14:47:47 Added task 266592cd-960d-4091-981c-8c25c44b10
Found task in queue: {266592cd-960d-4091-981c-8c25c44b1018 test-c
{"status":"Pulling from strm/helloworld-http","id":"latest"}
{"status":"Digest: sha256:bd44b0ca80c26b5eba984bf498a9c3bab0eb1c5
{"status":"Status: Image is up to date for strm/helloworld-http:l
2021/11/05 14:47:53 Sleeping for 10 seconds.
```

Great! At this point, we've created a task by calling the worker API's `POST /tasks` route. Now, when we make a `GET` request to `/tasks`, instead of seeing an empty list we should see output like this:

```
$ curl -v localhost:5555/tasks
*   Trying 127.0.0.1:5555...
* Connected to localhost (127.0.0.1) port 5555 (#0)
> GET /tasks HTTP/1.1
> Host: localhost:5555
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Fri, 05 Nov 2021 19:17:55 GMT
< Content-Length: 346
<
[
  {
    "ID":"266592cd-960d-4091-981c-8c25c44b1018",
    "ContainerID": "6df4e15a5c840b0ece1aede5378e344fb672c25161961
    "Name":"test-chapter-5-1",
    "State":2,
    "Image":"strm/helloworld-http",
    "Memory":0,
    "Disk":0,
    "ExposedPorts":null,
    "PortBindings":null,
    "RestartPolicy":"",
    "StartTime":"0001-01-01T00:00:00Z",
    "FinishTime":"0001-01-01T00:00:00Z"
  }
]
```

Also, we should see a container running on our local machine, which we can verify using the `docker ps` like so:

```
$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}\t{{.
```

```
CONTAINER ID   IMAGE                   STATUS          NAMES
6df4e15a5c84   strm/helloworld-http   Up 35 minutes   test-chapte
```

So far, we've queried the worker API to get a list of tasks by making a `GET` request to the `/tasks` route. Seeing the worker didn't have any, we created one by making a `POST` request to the `/tasks` route. Upon querying the API again by making a subsequent `GET` request `/tasks`, we got back a list containing our task.

Now, let's exercise the last bit of the worker's API functionality and stop our task. We can do this by making a `DELETE` request to the `/tasks/<taskID>` route, using the `ID` field from our previous `GET` request.

```
$ curl -v --request DELETE "localhost:5555/tasks/266592cd-960d-40
*   Trying 127.0.0.1:5555...
* Connected to localhost (127.0.0.1) port 5555 (#0)
> DELETE /tasks/266592cd-960d-4091-981c-8c25c44b1018 HTTP/1.1
> Host: localhost:5555
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 204 No Content
< Date: Fri, 05 Nov 2021 19:25:47 GMT
```

In addition to seeing our request received a `HTTP/1.1 204 No Content` response, we should see log output from the worker API that looks like the following:

```
2021/11/05 15:25:47 Added task 266592cd-960d-4091-981c-8c25c44b10
Found task in queue: {266592cd-960d-4091-981c-8c25c44b1018 6df4e1
2021/11/05 15:25:54 Attempting to stop container 6df4e15a5c840b0e
2021/11/05 15:26:05 Stopped and removed container 6df4e15a5c840b0
2021/11/05 15:26:05 Sleeping for 10 seconds.
```

We can confirm it's been stopped by checking the output of `docker ps` again:

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS
```

We can also confirm it by querying the API and checking the state of our task. In the response to our `GET /tasks` request, we should see the `State` of

the task is 3.

```
$ curl -v localhost:5555/tasks
*   Trying 127.0.0.1:5555...
* Connected to localhost (127.0.0.1) port 5555 (#0)
> GET /tasks HTTP/1.1
> Host: localhost:5555
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Fri, 05 Nov 2021 19:31:36 GMT
< Content-Length: 356
<
[
  {
    "ID":"266592cd-960d-4091-981c-8c25c44b1018",
    "ContainerID": "20d50c12fb2243f96183b81c00942a123cd2a48e463cc
    "Name":"test-chapter-5-1",
    "State":3,
    "Image":"strm/helloworld-http",
    "Memory":0,
    "Disk":0,
    "ExposedPorts":null,
    "PortBindings":null,
    "RestartPolicy":"",
    "StartTime":"0001-01-01T00:00:00Z",
    "FinishTime":"2021-11-05T19:30:04.661208966Z"
  }
]
```

5.7 Summary

- The API wraps the worker's functionality and exposes it as a HTTP server, thus making it accessible over a network. This strategy of exposing the worker's functionality as a web API will allow the manager to start and stop tasks, as well as query the state of tasks, across 1 or more workers.
- The API is made up of handlers, routes, and a router. *Handlers* are functions that accept a request and know how to process it and return a response. *Routes* are patterns that can be used to match the URL of

incoming requests (e.g. `/tasks`). And, finally, a *router* is the glue that makes it all work by

- The API uses the standard HTTP methods like GET, POST, DELETE to define the operations that will occur for a given route. For example, calling `GET /tasks` will return a list of tasks from the worker.
- While the API wraps the Worker's functionality, it does not interact with that functionality itself. Instead, it simply performs some administrative work and then places the task on the Worker's queue.

6 Metrics

This chapter covers

- Explaining why the worker needs to collect metrics
- Defining the metrics
- Creating a process to collect metrics
- Implementing a handler on the existing API

Imagine you're the host at a busy restaurant on a Friday night. You have six servers waiting on customers sitting at tables spread across the room. Each customer at each of those tables has different requirements. One customer might be there to have drinks and appetizers with a group of friends she hasn't seen in a while. Another customer might be there for a full dinner, complete with an appetizer and desert. Yet another customer might have strict dietary requirements and only eats plant-based food.

Now, a new customer walks in. It's a family of four: two adults, two teenage children. Where do you seat them? Do you place them at the table in the section being served by John, who already has three tables with four customers each? Do you place them at the table in Jill's section, who has six tables with a single customer each? Or, do you place them in Willie's section, who has a single table with three customers?

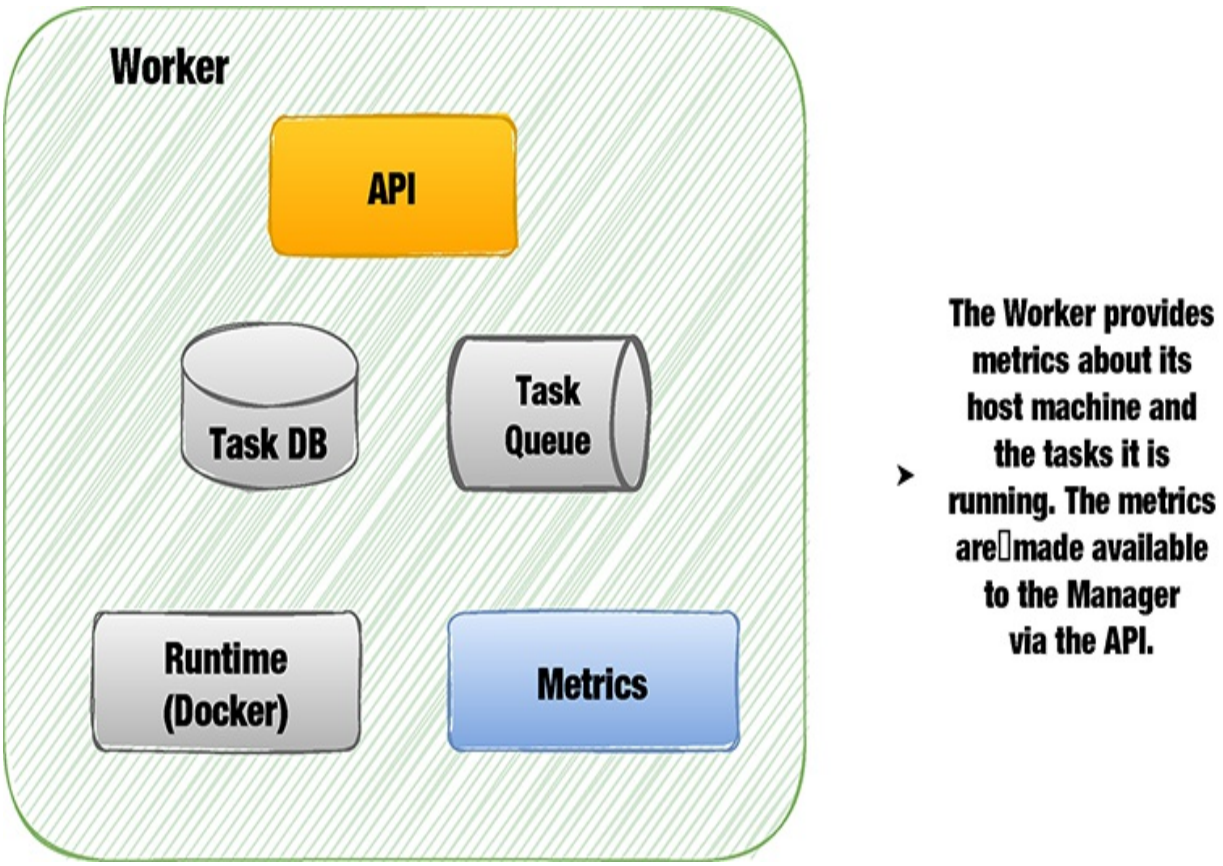
This scenario is exactly what the manager in an orchestration system deals with. Instead of six servers waiting on tables, you have six machines. Instead of customers, you have tasks, and instead of being hungry and wanting food and drinks the tasks want computing resources like CPU, memory, and disk. The manager's job in an orchestration system, like the host in the restaurant, is to place incoming tasks on the "best" worker machine that can meet the task's resource needs.

In order for the manager to do its job, however, it needs metrics that reflect how much work a worker is already doing. Those metrics are provided by the worker.

6.1 What metrics should we collect?

Before we dive deeper into metrics, it's good to refresh our memory about the worker and its components at a higher level. As you can see in figure 6.1, the two components that we'll be dealing with are the `API` and `Metrics`. In previous chapters, we covered the `Task DB`, `Task Queue`, and `Runtime` components. In the last chapter, we covered the API, which resulted in building an API server that wraps the Worker's lower level operations for starting and stopping tasks. Now, we want to dig deeper in the `Metrics` component, which will expose metrics on the same API.

Figure 6.1. Remembering the big picture of the worker's components.



In order for a worker to tell a manager how much work it's currently doing, what metrics will paint a reasonably accurate picture? Remember, we're not building a production-ready orchestrator. Systems like Borg, Kubernetes, and Nomad will have better metrics, both in quantity and quality, than will our

system. That's okay. We're trying to understand how an orchestration system works at a fundamental level, not replace existing systems.

In thinking about these metrics, let's look back at listing 3.6, in which we defined the `Task` struct. There are three fields in that struct that are relevant to this discussion: `CPU`, `Memory` and `Disk`. These fields represent how much CPU, memory, and disk space a task needs to perform its work. The values will be specified by humans like you and me when we submit our tasks to the system. If our task will be doing a lot of heavy computation, maybe it will need lots of CPU and memory. If our task uses a Docker image that is particularly large for some reason, we may want to specify an amount that provides a little overhead so there is room for the worker to download the image when it starts the task.

```
type Task struct {
    // prior fields not listed

    Cpu             float64
    Memory          int
    Disk            int

    // following fields not listed
}
```

If these are the resources that a user will specify when it submits a task to the system, then it makes sense that we should collect metrics about these resources from each of the workers. In particular, we're interested in metrics about the following:

- CPU usage (as a percentage)
- Total memory
- Available memory
- Total disk
- Available disk

6.2 Metrics available from the /proc filesystem

Now that we've identified the metrics we want to collect, let's talk about how we're going to collect them. On Linux systems, there is a pseudo-filesystem

named `/proc` which contains a range of information about the state of the system. A deep discussion about the `/proc` filesystem is beyond the scope of this book, and if you're interested in more details there are good sources that cover the topic. For our purposes, it's enough to understand that `/proc` is a special filesystem that is part of the Linux operating system and holds a wealth of information, including information about the state of the system's CPU, memory, and disk resources.

More info about /proc

If you're interested in more information about the /proc filesystem, there are many resources available on the web. Here are a couple to get started:

https://tldp.org/LDP/sag/html/proc-fs.html http://mng.bz/51MD

The nice thing about `/proc` is it appears like any other filesystem, which means users can interact with it the same way they interact with other "normal" filesystems. Items in `/proc` appear as files, which means standard tools like `ls` and `cat` can be used on them.

The files in the `/proc` filesystem that we're going to work with are:

- `/proc/stat` contains information about processes running on the system
- `/proc/meminfo` contains information about memory usage
- `/proc/loadavg` contains information about the system's load average

These files are the source of data that you see in many Linux commands like `ps`, `stat`, and `top`.

To get a sense of the data contained in these files, use the `cat` command to poke around them. For example, running the command `cat /proc/stat` on my laptop (which is running Manjaro Linux), I see a bunch of data about each of my CPUs. According to the `proc` man page (see `man 5 proc`), each line contains 10 values which represent the amount of time spent in various states. Those states are:

- **user** Time spent in user mode
- **nice** Time spent in user mode with low priority (nice)

- **system** Time spent in system mode
- **idle** Time spent in the idle task
- **iowait** Time waiting for I/O to complete
- **irg** Time servicing interrupts
- **softirq** Time servicing softirqs
- **steal** Stolen time, which is the time spent in other operating systems when running in a virtualized environment
- **guest** Time spent running a virtual CPU for guest operating systems under the control of the Linux kernel
- **guest_nice** Time spent running a niced guest

Listing 6.1. Using cat to look at the /proc/stat file

```
$ cat /proc/stat
cpu  724661 181 374910 105390121 4468 59434 23083 0 0 0  #1
cpu0 59580 33 29642 8786508 191 3560 2244 0 0 0  #2
cpu1 60502 3 31300 8779359 150 9016 2729 0 0 0
cpu2 58574 7 32002 8785331 139 3688 3159 0 0 0
cpu3 59564 9 30935 8787000 137 3259 2017 0 0 0
cpu4 59555 6 29208 8786670 369 3312 1950 0 0 0
cpu5 63148 16 37486 8755311 430 16914 2993 0 0 0
cpu6 60653 76 31349 8780196 483 4168 2040 0 0 0
cpu7 62622 2 33386 8781129 533 3402 1325 0 0 0
cpu8 60286 1 31729 8783928 542 3175 1219 0 0 0
cpu9 59229 2 29664 8787395 571 3038 1118 0 0 0
cpu10 59550 1 28925 8789436 468 2945 1100 0 0 0
cpu11 61392 18 29278 8787854 449 2952 1184 0 0 0
```

Running the command `cat /proc/meminfo` shows data about the memory being used by my system. This data is used by commands like `free`. For our purposes, we will focus on two values provided from `/proc/meminfo` (see `/proc/meminfo` in `man 5 proc` for more details).

- **MemTotal** Total usable RAM (i.e., physical RAM minus a few reserved bits and the kernel binary code)
- **MemAvailable** An estimate of how much memory is available for starting new applications, without swapping.

Listing 6.2. Using cat to look at the /proc/meminfo file

```
$ cat /proc/meminfo
```

```
MemTotal:       32488372 kB
MemFree:        21697264 kB
MemAvailable:   25975132 kB
Buffers:          512724 kB
Cached:          5829084 kB
SwapCached:            0 kB
Active:          1978056 kB
Inactive:        6165696 kB
Active(anon):      18368 kB
Inactive(anon):  3766080 kB
Active(file):    1959688 kB
Inactive(file):  2399616 kB
Unevictable:     1836208 kB
Mlocked:              32 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
[additional data truncated]
```

Running the command `cat /proc/loadavg` shows data about the system's load average. The first three fields in the output should look familiar, as they are the same as what you see when you run the `uptime` command. These numbers represent the number of jobs in the run queue (state `R`) or waiting for disk I/O averaged over 1, 5, and 15 minutes. The fourth field contains two values separated by a slash (i.e. this is not a fraction): the first value is the number of currently runnable kernel processes or threads, and the second value is the number of kernel processes and threads that currently exist on the system. (see `/proc/loadavg` in `man 5 proc`).

Listing 6.3. Using cat to look at the /proc/loadavg file

```
$ cat /proc/loadavg
0.14 0.16 0.18 1/1787 176550
```

While we will use the `/proc` filesystem for CPU and memory metrics, we won't use it to gather disk metrics. We could indeed use it, for there is the `/proc/diskstats` file. But, we're going to collect disk metrics a different way, which we'll talk more about here in a minute.

Since the metrics we're interested in are available from the `/proc` filesystem, we could write our own code to interact with `/proc` and pull out the data we need. We're not, however, going to take that route. Instead, we're going to

use a third-party library called `goprocinfo`.

6.3 Collecting metrics with goprocinfo

The goprocinfo library provides a range of types that allow us to interact with the `/proc` filesystem. Four our purposes we're going to focus on four types that will greatly simplify our work. They are:

- LoadAvg which provides the `ReadLoadAvg()` method and makes available the data from `/proc/loadavg`
- CpuStat which provides the `ReadStat()` method and makes available the data from `/proc/stat`
- MemInfo which provides the `ReadMemInfo()` method and makes available the data from `/proc/meminfo`
- Disk which provides the `ReadDisk()` method and makes available disk-related data using the `syscall` package from Go's standard library

We won't use every piece of data contained in each of these types, as we will see shortly.

In order to make it easy to use these metrics, let's create a wrapper around them. We'll start by adding the `Stats` struct you see in listing 6.1 below. This wrapper type contains five fields that will provide us everything we need. The `MemStats` field will hold all the memory-related data we need and will be a pointer to the `MemInfo` type from `goprocinfo`. The `DiskStats` field will hold all the necessary disk-related data and will be a pointer to `goprocinfo's` `` `Disk `` type. The `CpuStats` field will contain all the cpu-related data and will be a pointer to `goprocinfo's` `` `CPUStat `` type. Finally, the `LoadLoadStats` field will hold the relevant load-related data and will be a pointer to `goprocinf's` `` `LoadAvg `` type.

Listing 6.4. The `Stats` type we'll use to hold all the worker's metrics.

```
type Stats struct {
    MemStats  *linux.MemInfo
        DiskStats *linux.Disk
        CpuStats  *linux.CPUStat
        LoadStats *linux.LoadAvg
```

```
}
```

Now that we have defined our `Stats` type, let's take a step back and think about the kinds of metrics that might be useful. Starting with memory, what might we be interested in? It'd be good to know how much total memory the worker has. Knowing how much memory is available for new programs would also probably be useful. It'd also be good to know how much memory is being used. Similarly, it'd be useful to know how much memory is being used as a percentage of total memory.

With these memory metrics identified, let's add some helper methods to the `Stats` type that will make it quick and easy to get this data. We'll start with a method named `MemTotalKb`, seen in listing 6.2 below. This method simply returns the value of the `MemStats.MemTotal` field. We add the suffix `Kb` to the method name as a quick reminder of the units being used.

Listing 6.5. The `MemTotalKb()` helper method.

```
func (s *Stats) MemTotalKb() uint64 {
        return s.MemStats.MemTotal
}
```

Next, let's add the `MemAvailableKb` method seen in listing 6.3. Like `MemTotalKb`, it simply returns the value from a field in the `MemStats` field, in this case `MemAvailable`.

Listing 6.6. The `MemAvailableKb()` helper method.

```
func (s *Stats) MemAvailableKb() uint64 {
        return s.MemStats.MemAvailable
}
```

The `MemTotalKb` and `MemAvailable` methods let us figure out the last two memory-related metrics we identified, how much memory is being used as an absolute value and as a percentage of total memory. These metrics are provided by the `MemUsedKb` and `MemUsedPercent` methods in listing 6.4.

Listing 6.7. The `MemUsedKb()` and `MemUsedPercent()` helper methods.

```
func (s *Stats) MemUsedKb() uint64 {
```

```
        return s.MemStats.MemTotal - s.MemStats.MemAvailable
}

func (s *Stats) MemUsedPercent() uint64 {
        return s.MemStats.MemAvailable / s.MemStats.MemTotal
}
```

Now let's turn our attention to disk-related metrics. Similar to our memory metrics, it'd be good to know how much total disk is available on a worker machine, how much is free, and how much is being used. Unlike the memory-related methods, our disk-related methods won't need to perform any calculations. The data is provided to us directly from `goprocinfo's` `Disk` type. So, let's create the `DiskTotal`, `DiskFree`, and `DiskUsed` methods seen below in listing 6.5.

Listing 6.8. The helper methods for collecting metrics about the amount of total disk space, the amount of free disk space, and the amount of disk space used.

```
func (s *Stats) DiskTotal() uint64 {
        return s.DiskStats.All
}

func (s *Stats) DiskFree() uint64 {
        return s.DiskStats.Free
}

func (s *Stats) DiskUsed() uint64 {
        return s.DiskStats.Used
}
```

Finally, let's talk about CPU-related metrics. The two mostly commonly used metrics when we talk about CPU-related metrics are *load average* and *usage*. As we mentioned earlier, *load average* can be seen in the output of the `uptime` command, which come from `/proc/loadavg`.

```
$ uptime
 14:38:18 up 6 days, 22:39,  2 users,  load average: 0.43, 0.32,

$ cat /proc/loadavg
0.43 0.32 0.33 1/2462 865995
```

For CPU *usage*, however, the story is slightly more complicated. When we talk about "cpu usage", we typically talk in terms of percentages. For

example, currently on my laptop the cpu usage is 2%. But, what does that actually mean?

As we talked about previously, on a Linux operating system a CPU spends its time in various states. Moreover, we can see how much time our CPU(s) are spending in each of the states (`user`, `nice`, `system`, `idle`, etc.) by looking at `/proc/stat`. Knowing how much time our CPUs are spending in these individual states is nice, but it doesn't translate into that single percentage we use when we say, "cpu percentage is 2%".

Unfortunately, the `CPUStat` type provided by the goprocinfo library doesn't provide us with any useful helper methods to calculate the cpu usage; it simply provides us the `CPUStat` type as you can see below.

```
type CPUStat struct {
        Id        string `json:"id"`
        User      uint64 `json:"user"`
        Nice      uint64 `json:"nice"`
        System    uint64 `json:"system"`
        Idle      uint64 `json:"idle"`
        IOWait    uint64 `json:"iowait"`
        IRQ       uint64 `json:"irq"`
        SoftIRQ   uint64 `json:"softirq"`
        Steal     uint64 `json:"steal"`
        Guest     uint64 `json:"guest"`
        GuestNice uint64 `json:"guest_nice"`
}
```

So, it is up to us to calculate this percentage ourselves. Luckily, we don't have to do too much work, because this problem has been discussed in a StackOverflow post titled, Accurate calculation of CPU uage given in percentage in Linux. According to this post, the general algorithm for performing this calculation is this:

1. sum the values for the idle states
2. sum the values for the non-idle states
3. sum the total of idle and non-idle states
4. subtract the idle from the total and divide the result by the total

Thus, we can code this algorithm as you see in listing 6.6.

Listing 6.9. The `CpuUsage()` method encodes the algorithm that will give us the CPU usage as a percentage.

```
func (s *Stats) CpuUsage() float64 {

        idle := s.CpuStats.Idle + s.CpuStats.IOWait
        nonIdle := s.CpuStats.User + s.CpuStats.Nice + s.CpuStats
        total := idle + nonIdle


        if total == 0 {
                return 0.00
        }

        return (float64(total) - float64(idle)) / float64(total)
}
```

At this point, we have laid the foundation for gathering metrics that reflect the amount of work an individual worker is performing. All that is left now is to wrap up our work into a few functions that will return a fully populated `Stats` type that we can use in the worker's API.

The first of these functions is the `GetStats()` function seen in listing 6.7. This function sets the fields `MemStats`, `DiskStats`, `CpuStats`, `LoadStats` in the `Stats` struct by calling the appropriate helper functions.

Listing 6.10. The `GetStats()` function populates an instance of the `Stats` type and returns a pointer to the caller.

```
func GetStats() *Stats {
        return &Stats{
                MemStats:  GetMemoryInfo(),
                DiskStats: GetDiskInfo(),
                CpuStats:  GetCpuStats(),
                LoadStats: GetLoadAvg(),
        }
}
```

Each of the helper functions used in the `GetStats` function takes a similar format. It starts by calling the relevant function from the `goprocinfo` library. It then checks if any errors were returned from the function call. And, finally, it returns the data in the relevant struct.

It’s worth noting that if there is an error in calling the relevant `goprocinfo` function, we simply print an error message and return a pointer to the appropriate type, e.g. `&linux.MemInfo{}`. The returned type will be populated with the appropriate zero value (i.e. the empty string `""` for strings and `0` for numbers).

Listing 6.11. These helper functions returns metrics from the `/proc` filesystem, with the exception of the `GetDiskInfo()` function. Under the hood, it uses the `syscall` package from Go’s standard library.

```
func GetMemoryInfo() *linux.MemInfo {
        memstats, err := linux.ReadMemInfo("/proc/meminfo")
        if err != nil {
                log.Printf("Error reading from /proc/meminfo")
                return &linux.MemInfo{}
        }

        return memstats
}

// GetDiskInfo See https://godoc.org/github.com/c9s/goprocinfo/li
func GetDiskInfo() *linux.Disk {
        diskstats, err := linux.ReadDisk("/")
        if err != nil {
                log.Printf("Error reading from /")
                return &linux.Disk{}
        }

        return diskstats
}

// GetCpuInfo See https://godoc.org/github.com/c9s/goprocinfo/lin
func GetCpuStats() *linux.CPUStat {
        stats, err := linux.ReadStat("/proc/stat")
        if err != nil {
                log.Printf("Error reading from /proc/stat")
                return &linux.CPUStat{}
        }

        return &stats.CPUStatAll
}

// GetLoadAvg See https://godoc.org/github.com/c9s/goprocinfo/lin
func GetLoadAvg() *linux.LoadAvg {
        loadavg, err := linux.ReadLoadAvg("/proc/loadavg")
```

```
        if err != nil {
                log.Printf("Error reading from /proc/loadavg")
                return &linux.LoadAvg{}
        }

        return loadavg
}
```

6.4 Exposing the metrics on the API

Now that we've done all the hard work, there are only three things left to do in order to expose the worker's metrics on its API:

1. add a method to the worker to regularly collect metrics
2. add a handler method to the API
3. add a `/stats` route to the API

To regularly collect our metrics, let's add a method called `CollectStats` to our worker in the `worker.go` file. This method, seen in listing 6.9, uses an infinite loop, inside of which we call the `GetStats()` function we created earlier. Note that we also set the worker's `TaskCount` field. Finally, we sleep for fifteen seconds. Why sleep for fifteen seconds? This is an arbitrary decision that is mainly intended to slow down how frequently our system is performing actions so that we humans can observe what is going on. In a real production system, where users might be submitting tens, hundreds, or even thousands of tasks per minute, we'd want to collect metrics in a more real-time fashion.

Listing 6.12. The worker's new `CollectStats()` method.

```
func (w *Worker) CollectStats() {
        for {
                log.Println("Collecting stats")
                w.Stats = GetStats()
        w.TaskCount = w.Stats.TaskCount
                time.Sleep(15 * time.Second)
        }
}
```

Next, let's add the new handler method, called `GetStatsHandler`, to the API

in the `handlers.go` file. Like the other handlers we created in chapter 5, this one takes two arguments, an `http.ResponseWriter` named `w` and a pointer to an `http.Request` named `r`. The body of the method is pretty simple. It sets the `Content-Type` header to `application/json` to let the caller know the response contains JSON-encoded content. It then sets the response code to `200`. Finally, it encodes the worker's `Stats` field. Thus, `GetStatsHandler` is simply encoding and returning the metrics in the worker's `Stats` field, which gets refreshed every fifteen seconds by the `CollectStats` method above.

Listing 6.13. The API's new `GetStatsHandler()` method will be used for requests to the new `/stats` route created in listing 6.11.

```
func (a *Api) GetStatsHandler(w http.ResponseWriter, r *http.Requ
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(200)
        json.NewEncoder(w).Encode(a.Worker.Stats)
}
```

The last thing to do is to update the API's routes in the `api.go` file. Here, we will create a new route, `/stats`. That route will only support `GET` requests and will call the `GetStatsHandler` we created above.

Listing 6.14. Adding the new `/stats` route to the `api.go` file.

```
a.Router.Route("/stats", func(r chi.Router) {
    r.Get("/", a.GetStatsHandler)
})
```

Since we've added this new route to our API, let's update the route table from chapter 5 to provide a complete picture of what it now looks like.

Table 6.1. Our updated route table for the worker API.

Method	Route	Description	Request Body	Response Body	Status code
GET	/tasks	get a list of all tasks	none	list of tasks	200

POST	/tasks	create a task	JSON-encoded task.TaskEvent	none	201
DELETE	/tasks/{taskID}	stop the task identified by `taskID`	none	none	204
GET	/stats	get metrics about the worker	none	JSON-encoded stats.Stats	200

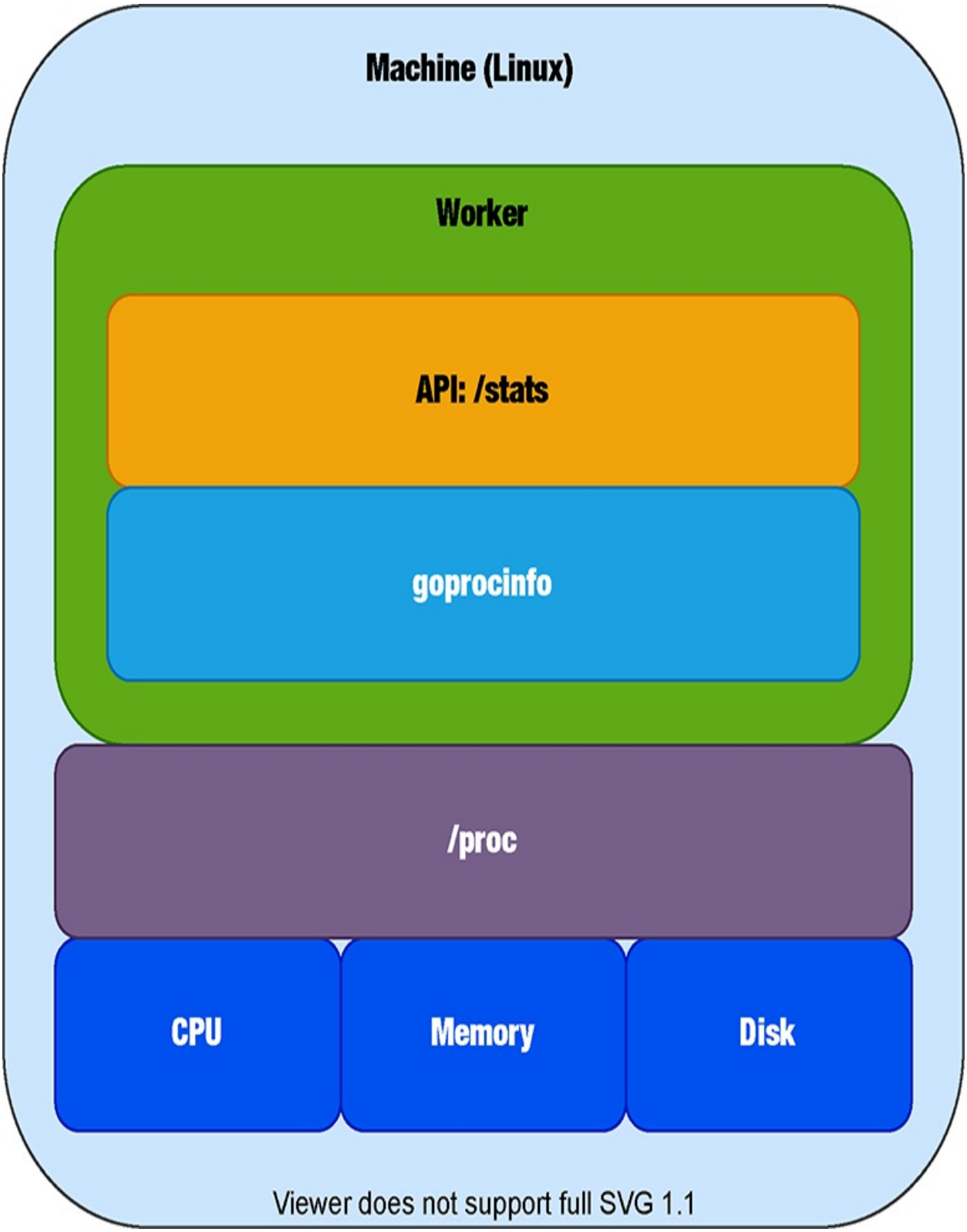
6.5 Putting it all together

Before we put this all together and take it for a test run, let's quickly review what we've done.

- We created a new file, `stats.go`
- In `stats.go` we created a new `Stats` type to hold the worker's metrics
- Also in `stats.go`, we created a `GetStats()` function that uses the `goprocinfo` library to collect metrics and populate the `Stats` type with data
- We added the `CollectStats` method to the worker, which will call the `GetStats()` function in an infinite loop
- We added the `GetStatsHandler` method to the worker's handlers in `handlers.go`
- We added a new route, `/stats`, to the worker's API

At a conceptual level, the above work looks like that in figure 6.2.

Figure 6.2. The worker runs on a Linux machine and serves an API that includes the `/stats` endpoint. The metrics served on the `/stats` endpoint are collected using the `goprocinfo` library, which interacts directly with the Linux `/proc` filesystem.



Now, to see our work in action we need to write a program, like we've done in past chapters, that will glue all of our work together. In this case, we can

actually re-use the same program we wrote in chapter 5. We only need to make one minor change.

So, open up the `main.go` program we used in chapter 5. In the `main()` function after the call to `runTasks`, add a call to the worker's new `CollectStats` method. And, like the `runTasks` method, execute that call to `CollectStats` in a separate goroutine.

Listing 6.15. Updating the `main()` function from our `main.go` file. To allow the API to provide metrics about the worker's state, we just add a line to run the `CollectStats()` method in a separate goroutine.

```
func main() {
        host := os.Getenv("CUBE_HOST")
        port, _ := strconv.Atoi(os.Getenv("CUBE_PORT"))

        fmt.Println("Starting Cube worker")

        w := worker.Worker{
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]*task.Task),
        }
        api := worker.Api{Address: host, Port: port, Worker: &w}

        go runTasks(&w)
        go w.CollectStats()
        api.Start()
}
```

After updating the `main.go` file, start up the API the same way you did in chapter 5. You'll notice that the API's log output shows that it's collecting stats every 15 seconds.

```
$ CUBE_HOST=localhost CUBE_PORT=5555 go run main.go
Starting Cube worker
2021/12/28 14:03:35 No tasks to process currently.
2021/12/28 14:03:35 Sleeping for 10 seconds.
2021/12/28 14:03:35 Collecting stats
2021/12/28 14:03:45 No tasks to process currently.
2021/12/28 14:03:45 Sleeping for 10 seconds.
2021/12/28 14:03:50 Collecting stats
```

Now that the API is running, query the new `/stats` endpoint from a different

terminal. You should see output about memory, disk and cpu usage.

```
$ curl localhost:5555/stats|jq .
{
  "MemStats": {
    "mem_total": 32488372,
    "mem_free": 14399056,
    "mem_available": 23306576,
    [....]
  },
  "DiskStats": {
    "all": 1006660349952,
    "used": 39346565120,
    "free": 967313784832,
    "freeInodes": 61645909
  },
  "CpuStats": {
    "id": "cpu",
    "user": 4819423,
    "nice": 701,
    "system": 2140212,
    "idle": 502094668,
    "iowait": 14448,
    "irq": 561115,
    "softirq": 178454,
    "steal": 0,
    "guest": 0,
    "guest_nice": 0
  },
  "LoadStats": {
    "last1min": 0.78,
    "last5min": 0.55,
    "last15min": 0.43,
    "process_running": 2,
    "process_total": 2336,
    "last_pid": 581117
  },
  "TaskCount": 0
}
```

6.6 Summary

- The worker exposes metrics about the state of the machine where it is running. The metrics—about CPU, memory, and disk usage—will be used by the manager to make scheduling decisions.

- To make gathering metrics easier, we use a third-party library called `goprocinfo`. This library handles most of the low-level work necessary to get metrics from the `/proc` filesystem.
- The metrics are made available on the same API that we built in chapter 5. Thus, the manager will have a uniform way to interact with workers: making HTTP calls to `/tasks` to perform task operations, and making calls to `/stats` to gather metrics about a worker's current state.

7 The manager enters the room

This chapter covers

- Reviewing the purpose of the manager
- Designing a naive scheduling algorithm
- Implementing the manager's methods for scheduling and updating tasks

In chapters 4, 5, and 6, we implemented the Worker component of Cube, our orchestration system. We focused on the core functionality of the Worker in chapter 4, which enabled the worker to start and stop tasks. In chapter 5, we added an API to the worker. This API wrapped the functionality we built in chapter 4 and made it available from standard HTTP clients (e.g. curl). And, finally, in chapter 6, we added the ability for our worker to collect metrics about itself and expose those on the same API. With this work, we can run multiple workers, with each worker running multiple tasks.

Now, we'll move our attention to the Manager component of Cube. As we mentioned in chapter 1, the manager is the brain of an orchestrator. While we have multiple workers, we wouldn't want to ask the users of our orchestration system to submit their tasks directly to a worker. Why? This would place an unnecessary burden on users, forcing them to be aware of how many workers existed, how many tasks they were already running, and then to pick one. Instead, we encapsulate all of that administrative work into the manager. The users submit their tasks to the manager, and it figures out which worker in the system can best handle the task.

Unlike Workers, the Cube orchestrator will have a single Manager. This is a practical design decision meant to simplify the number of issues we need to consider in our manager implementation.

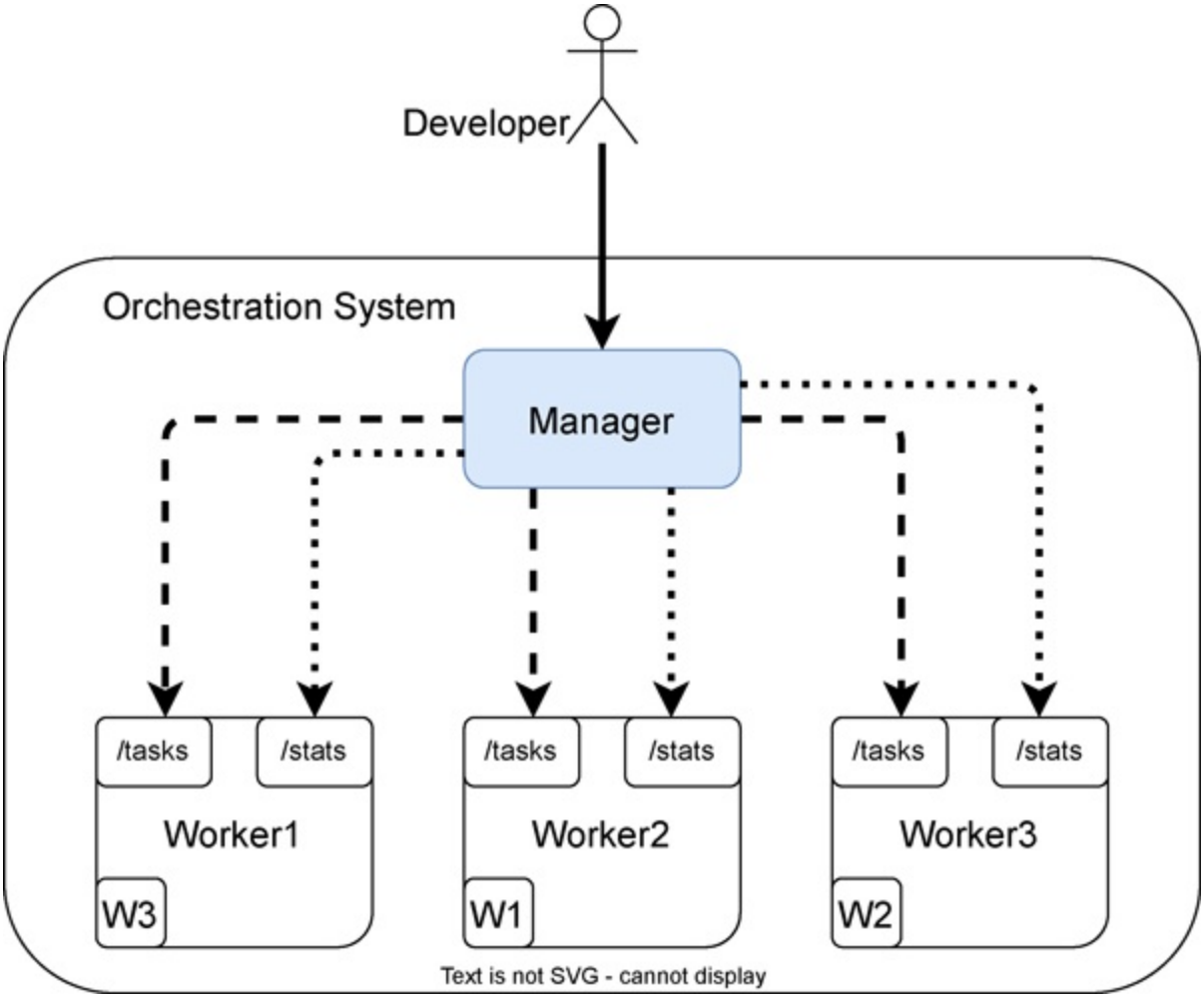
By the end of this chapter, we will have implemented a Manager than can submit tasks to Workers, using a naive round-robin scheduling algorithm.

7.1 The Cube manager

The manager component allows us to isolate administrative concerns from execution concerns. This is a design principle known as *separation of concerns*. *Administrative* concerns in an orchestration system include things like the following:

- Handling requests from users
- Assigning tasks to workers who are best able to perform them (i.e. scheduling)
- Keeping track of task and worker state
- Restarting failed tasks

Figure 7.1. The manager is responsible for administrative tasks, similar to the function of a restaurant host seating customers. It will use the worker's /tasks and /stats API endpoints to perform its administrative duties.



Every orchestration system has a manager component. Google's Borg calls it the *borgmaster*. Hashicorp's Nomad uses the unimaginative yet functional term *server*. Kubernetes doesn't have a singular name for this component, but instead specifically identifies the subcomponents (API server, controller manager, etcd, scheduler).

Control plane vs data plane

Another way to think of separation of concerns is the concept of *control plane* vs *data plane*. In the world of networking, you'll find these terms used frequently, and they refer to "plane of existence".

In a network, the control plane *controls* how data moves from point A to point B. This plane is responsible for things like creating routing tables, which are determined by different protocols (such as the Border Gateway Protocol, BGP, and the Open Shortest Path First, OSPF, protocol). This plane performs functions similar to the administrative concerns of our manager.

Unlike the control plane, the data plane does the actual work of moving the data around. This plane performs functions similar to the execution concerns of our worker.

7.1.1 The components that make up the manager

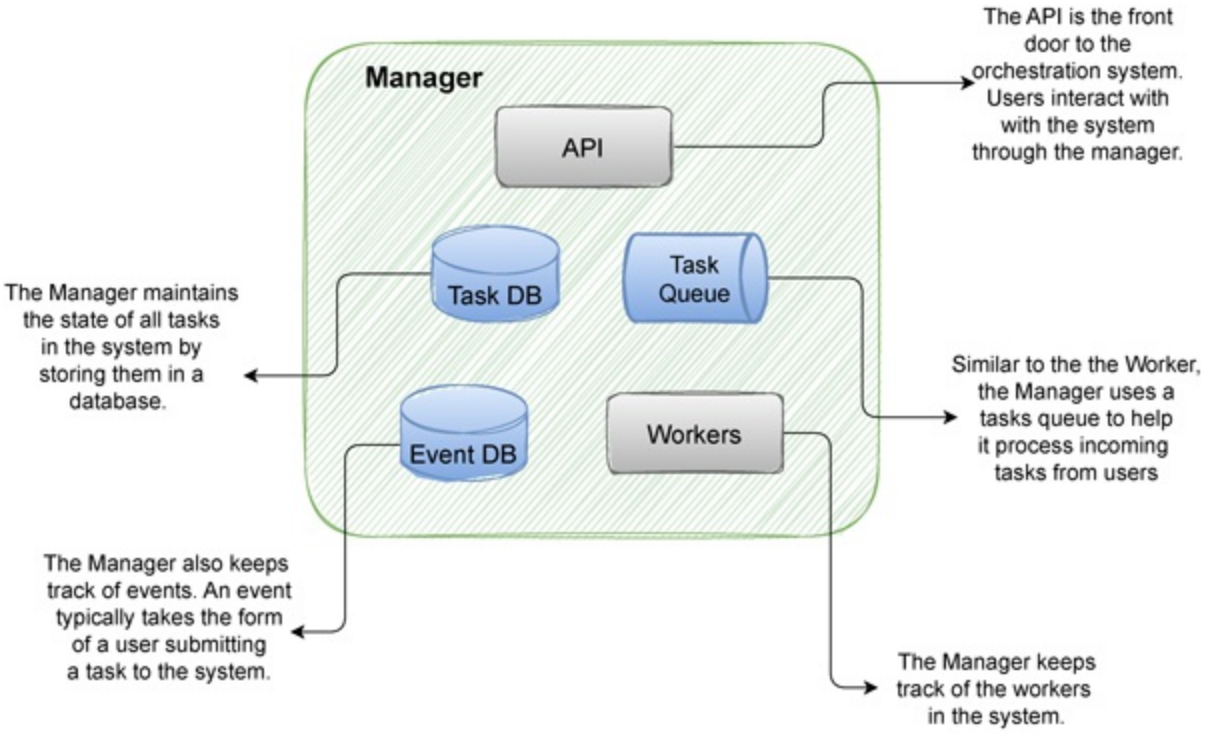
Like our worker, our manager will be comprised of several subcomponents, as seen in figure 7.2. The manager will have a `Task DB`, which, like the worker, will store tasks. In contrast to the worker's Task DB, however, the manager's will contain *all* tasks in the system.

The manager will also have an `Event DB`, which will store events (i.e., task.TaskEvent). This subcomponent is mostly a convenient way for us to separate metadata from task-specific data. Metadata includes things like the timestamp when a user submitted a task to the system, . We'll make use of it in a later chapter when we implement a CLI for the manager.

Like the worker, for the initial implementation of the manager, we're going to use an in-memory map to store tasks and events. The manager's `Workers` subcomponent is a list of the workers it manages. Like the worker, it will also

have a `Task Queue`. And, finally, the manager will have an `API`, similar to the worker. (As we did with the worker, we're going to address the manager's API in a separate chapter. So we'll defer further discussion of it until a later chapter.)

Figure 7.2. The manager's components are similar to the worker's, with the addition of an `Event DB` and a list of `Workers`.



With this foundation laid, we can move on to implementation.

7.2 The Manager struct

Like the worker, we created a skeleton of the manager implementation in chapter 2. At the core of that manager skeleton is the `Manager` struct, which will contain fields that represent the subcomponents identified above. You can see this struct in listing 7.1, which should be in the same state we left it in chapter 2.

Since it has been a while, let's remind ourselves of the requirements for our manager. In chapter 1, we identified these requirements:

1. Accept requests from users to start and stop tasks.
2. Schedule tasks onto worker machines.
3. Keep track of tasks, their states, and the machine on which they run.

If you need more of a reminder about the `Manager` struct and its field, please look back at section 2.3 in chapter 2.

Listing 7.1. The Manager struct.

```
package manager

import (
        "bytes"
        "cube/task"
        "cube/worker"
        "encoding/json"
        "fmt"
        "log"
        "net/http"

        "github.com/golang-collections/collections/queue"
        "github.com/google/uuid"
)

type Manager struct {
        Pending       queue.Queue
        TaskDb        map[uuid.UUID]*task.Task
        EventDb       map[uuid.UUID]*task.TaskEvent
        Workers       []string
        WorkerTaskMap map[string][]uuid.UUID
        TaskWorkerMap map[uuid.UUID]string
}
```

7.3 Implementing the manager's methods

Now that we've reminded ourselves of what the `Manager` struct looks like, let's move forward and remember what skeleton methods we had previously defined on the struct.

Listing 7.2. The stubbed out versions of the Manager's `SelectWorker`, `UpdateTasks`, and `SendWork` methods.

```
func (m *Manager) SelectWorker() {
    fmt.Println("I will select an appropriate worker")
}

func (m *Manager) UpdateTasks() {
    fmt.Println("I will update tasks")
}

func (m *Manager) SendWork() {
    fmt.Println("I will send work to workers")
}
```

We're going to implement these methods in the following order:

- SelectWorker
- SendWork
- UpdateTasks

7.3.1 Implementing the SelectWorker method

The `SelectWorker` method will serve as the scheduler in this early phase of implementing our manager. It's sole purpose will be to pick one of the workers from the manager's list of workers, i.e. the `Workers` field, which is a slice of strings. We're going to start with a naive round-robin scheduling algorithm that begins by simply selecting the first worker from the list of `Workers` and storing it in a variable. From this point forward, the algorithm looks like so:

1. Check if we are at the end of the `Workers` list
2. If we are not, select the next worker in the list
3. Else, return to the beginning and select the first worker in the list

In order to implement this algorithm, we need to make a minor change to the `Manager` struct. As you can see in listing 7.3, we've added the field `LastWorker`. We'll use this field to store an integer, which will be an index into the `Workers` slice, thus giving us a worker.

Listing 7.3. Adding the `LastWorker` field to the Manager struct.

```
type Manager struct {
```

```
    // previous fields omitted
        LastWorker    int
```

Let's move on now to the actual scheduling algorithm. As you can see in listing 7.4, it's only nine lines of code (not counting the method signature). We start the process by declaring the variable `newWorker`, which represents the lucky worker that has been chosen to run a task. Then, we use an `if/else` block to actually choose the worker. In this block, we first check if the worker we chose during the last run is the last worker in our list of workers. If not, then we set `newWorker` to the next worker in the list of workers, and we increment the value of `LastWorker` by `1`. If the previous worker chosen is the last one in our list, then we start over from the beginning, choosing the first worker in the list, and setting `LastWorker` accordingly. Finally, we return the worker to the caller.

Listing 7.4. The `SelectWorker` method implements an extremely naive scheduling algorithm.

```
func (m *Manager) SelectWorker() string {
        var newWorker int  #1
        if m.LastWorker+1 < len(m.Workers) { #2
                newWorker = m.LastWorker + 1 #3
                m.LastWorker++ #4
        } else { #5
                newWorker = 0 #6
                m.LastWorker = 0 #7
        }

        return m.Workers[newWorker] #8
}
```

It's worth taking a moment to talk about the format of the strings stored in the manager's `Workers` field. The field itself is of type `[]string`, so technically the value of the strings could be anything. In practice, however, these are going to take the form of `<hostname>:<port>`. If you recall from chapters 5 and 6, when we started the worker's API, we specified the `CUBE_HOST` and `CUBE_PORT` environment variables. The former we set to `localhost`, and the latter we set to `5555`. So, the manager's `Workers` field contains a list of `<hostname>:<port>` values, which specifies the address where the worker's API is running.

7.3.2 Implementing the SendWork method

The next method we need to implement is the manager's `SendWork` method. It is the workhorse of the manager and performs the following process:

1. Check if there are task events in the `Pending` queue
2. If there are, select a worker to run a task
3. Pull a task event off the pending queue
4. Set the state of the task to `Scheduled`
5. Perform some administrative work that makes it easy for the manager to keep track of which workers tasks are running on
6. JSON encode the task event
7. Send the task event to the selected worker
8. Check the response from the worker

Let's implement steps 1-6 in the above process with the code in listing 7.5. We use an if/else block that sets up our conditional flow: it checks the length of the manager's `Pending` queue, and if the length is greater than zero—meaning there are tasks to process—it moves on to the next steps.

Listing 7.5. The manager's `SendWork` method.

```
func (m *Manager) SendWork() {
        if m.Pending.Len() > 0 {
                w := m.SelectWorker() #1

                e := m.Pending.Dequeue()
                te := e.(task.TaskEvent) #2
                t := te.Task
                log.Printf("Pulled %v off pending queue", t)

                m.EventDb[te.ID] = &te #3
                m.WorkerTaskMap[w] = append(m.WorkerTaskMap[w], t
                m.TaskWorkerMap[t.ID] = w #5

                t.State = task.Scheduled
                m.TaskDb[t.ID] = &t

                data, err := json.Marshal(te) #6
                if err != nil {
                        log.Printf("Unable to marshal task object
                }
```

Once the `SendWork` method has pulled a task of the `Pending` queue and encoded it as JSON, all that's left to do is send the task to the selected worker. These final two steps are implemented in listing 7.6. Sending the task to the worker involves building the URL using the worker's `host` and `port`, which we got when we called the manager's `SelectWorker` method above. From there, we use the `Post` function from the `net/http` package in the standard library. Then we decode the response body and print it. Notice that we're also checking errors along the way.

```
                url := fmt.Sprintf("http://%s/tasks", w)
                 resp, err := http.Post(url, "application/json", b
                 if err != nil {
                         log.Printf("Error connecting to %v: %v",
                         m.Pending.Enqueue(t)
                         return
                 }

                 d := json.NewDecoder(resp.Body)
                 if resp.StatusCode != http.StatusCreated {
                         e := worker.ErrResponse{}
                         err := d.Decode(&e)
                         if err != nil {
                                 fmt.Printf("Error decoding respon
                                 return
                         }
                         log.Printf("Response error (%d): %s", e.H
                         return
                 }

                 t = task.Task{}
                 err = d.Decode(&t)
                 if err != nil {
                         fmt.Printf("Error decoding response: %s\n
                         return
                 }
                 log.Printf("%#v\n", t)
         } else {
                 log.Println("No work in the queue")
         }
}
```

As you can see from the above code, the manager is interacting with the worker via the worker API we implemented in chapter 5.

7.3.3 Implementing the UpdateTasks method

At this point, our manager can select a worker to run a task and then send that task to the selected worker. It has also stored the task in its `TaskDB` and `EventsDB` databases. But, the manager's view of the task is only from its own perspective. Sure, it sent it to the worker and, if all went well, the worker responded with `http.StatusCreated` (i.e. `201`). But, even if we receive a `http.StatusCreated` response, that just tells the us that the worker received the task and added it to its queue. This response gives us no indication that the task was started successfully and is currently running. What if the task failed when the worker attempted to start it? How might a task fail, you ask? Here are a few:

- The user specified a non-existent Docker image, so that when the worker attempts to start the task Docker complains that it can't find the image
- The worker's disk is full, so it doesn't have enough space to download the Docker image
- There is a bug in the application running inside the Docker container that prevents it from starting correctly (maybe the creator of the container image left out an important environment variable that the application needs to start up properly)

These are just several ways a task might fail. While the manager doesn't necessarily need to know about every possible way a task might fail, what it does need to do is check in with the worker periodically to get status updates on the tasks its running. Moreover, the manager needs to get status updates for every worker in the cluster.

With this in mind, let's implement the manager's `UpdateTasks` method.

The general shape of updating tasks from each worker is straightforward. For each worker, we perform the following steps:

1. Query the worker to get a list of its tasks
2. For each task, update its state in the manager's database so it matches the state from the worker

The first step can be seen in listing 7.6. It should look familiar by this point. The manager starts by making a `GET /tasks` HTTP request to a worker using the `Get` method from the `net/http` package. It checks if there were any errors, such as connection issues (maybe the worker was down for some reason). If the manager was able to connect to the worker, it then checks the response code and ensures it received a `http.StatusOK` (i.e. `200`) response code. Finally, it decodes the JSON data in the body of the response, which should result in a list of the worker's tasks.

Listing 7.6. TODO

```
func (m *Manager) UpdateTasks() {
        for _, worker := range m.Workers {
                log.Printf("Checking worker %v for task updates",
                url := fmt.Sprintf("http://%s/tasks", worker)
                resp, err := http.Get(url)
                if err != nil {
                        log.Printf("Error connecting to %v: %v",
                }

                if resp.StatusCode != http.StatusOK {
                        log.Printf("Error sending request: %v", e
                }

                d := json.NewDecoder(resp.Body)
                var tasks []*task.Task
                err = d.Decode(&tasks)
                if err != nil {
                        log.Printf("Error unmarshalling tasks: %s
                }
}
```

The second step, seen below in listing 7.7, runs inside the main for loop from listing 7.6. There is nothing particularly sophisticated or clever about how we're updating the tasks. We start by checking if the task's state in the manager is the same as that of the worker; if not, we set the state to the state reported by the worker. (In this way, the manager treats the workers as the authoritative source for the *state of the world*, that is, the current state of the tasks in the system.) Once we've updated the task's state, we then update its `StartTime` and `FinishTime`. And, finally, we update the task's `ContainerID`.

Listing 7.7. TODO

```
            for _, t := range tasks {
                    log.Printf("Attempting to update task %v"

                    _, ok := m.TaskDb[t.ID]
                    if !ok {
                            log.Printf("Task with ID %s not f
                            return
                    }

                    if m.TaskDb[t.ID].State != t.State {
                            m.TaskDb[t.ID].State = t.State
                    }

                    m.TaskDb[t.ID].StartTime = t.StartTime
                    m.TaskDb[t.ID].FinishTime = t.FinishTime
                    m.TaskDb[t.ID].ContainerID = t.ContainerI
                }
}
```

With the implementation of the `UpdateTasks` method, we have now completed the core functionality of our manager. So let's quickly summarize what we've accomplished thus far before continuing:

- With the `SelectWorker` method we've implemented a simple, but naive, scheduling algorithm to assign tasks to workers
- With the `SendWork` method we've implemented a process that uses the `SelectWorker` method and sends individual tasks to assigned workers via that worker's API
- With the `UpdateTasks` method we've implemented a process for the manager to update it's view of the state of all the tasks in the system

That's a large chunk of work we've just completed. Take a moment to celebrate your achievement before moving on to the next section!

7.3.4 Adding a task to the manager

While we have implemented the core functionality of the manager, there are a couple more methods that we still need to implement. The first of these methods is the `AddTask` method seen in listing 7.7. This method should look familiar, as its similar to the `AddTask` method we created in the worker. It also serves a similar purpose: it's how tasks are added to the manager's queue.

Listing 7.8. The manager's `AddTask` method serves the same purpose as the worker's: adding a task to the manager's queue of pending tasks.

```
func (m *Manager) AddTask(te task.TaskEvent) {
        m.Pending.Enqueue(te)
}
```

7.3.5 Creating a manager

Finally, let's create the `New` function seen in listing 7.8. This is a helper function that takes in a list of workers, creates an instance of the manager, and returns a pointer to it. The bulk of the work performed by this function is initializing the necessary subcomponents used by the manager. It sets up the `taskDB` and `eventDb` databases. Next, it initializes the `workerTaskMap` and `taskWorkerMap` maps that help the manager more easily identify where tasks are running. While this function isn't technically called a *constructor*, as in some other object-oriented languages, it performs a similar function and will be used in the process of starting the manager.

Listing 7.9. TODO

```
func New(workers []string) *Manager {
        taskDb := make(map[uuid.UUID]*task.Task)
        eventDb := make(map[uuid.UUID]*task.TaskEvent)
        workerTaskMap := make(map[string][]uuid.UUID)
        taskWorkerMap := make(map[uuid.UUID]string)
        for worker := range workers {
                workerTaskMap[workers[worker]] = []uuid.UUID{}
        }

        return &Manager{
                Pending:       *queue.New(),
                Workers:       workers,
                TaskDb:        taskDb,
                EventDb:       eventDb,
                WorkerTaskMap: workerTaskMap,
                TaskWorkerMap: taskWorkerMap,
        }
}
```

With the addition of the `AddTask` method and `New` function, we've completed the initial implementation of the Cube manager. Now, all that's left to do is

take it for a spin!

7.4 An interlude on failures and resiliency

It's worth pausing here for a moment to identify a weakness in our implementation. While the manager can select a worker from a pool of workers and send it a task to run, it is not dealing with failures. The manager is simply recording the state of the world in its `Task DB`.

What we are building towards, however, is a declarative system where a user declares the desired state of a task. The job of the manager is to honor that request by making a reasonable effort to bring the task into the declared state. For now, a *reasonable effort* means making a single attempt to bring the task into the desired state. We are going to revisit this topic in a later chapter, where we will consider additional steps the manager can take in the face of failures in order to build a more resilient system.

7.5 Putting it all together

By now, the pattern were using to build Cube should be clear. We spend the bulk of the chapter writing the core pieces we need, then at the end of the chapter we write, or update, a `main()` function in our project's `main.go` file to make use of our work. We'll continue using this same pattern here and for the next few chapters.

For this chapter, we're going to start by using the `main.go` file from chapter 6 as our starting point. Whereas in past chapters we focused exclusively on running a worker, now we want to run a worker and a manager. We want to run them both because running a manager in isolation makes no sense: the need for a manager only makes sense in the context of having one or more workers.

Let's make a copy of the `main.go` file from chapter 6. As we said, this is our starting point for running the worker and the manager. Our previous work already knows how to start an instance of the worker. As we can see in listing 7.9, we create an instance of the worker, `w`, and then we create an instance of

the worker API, `api`. Next, we call the `runTasks` function, also defined in `main.go`, and pass it a pointer to our worker `w`. We make the call to `runTasks` using a goroutine, identified by the `go` keyword before the function call. Similarly, we use a second goroutine to call the worker's `CollectStats()` method, which periodically collects stats from the machine where the worker is running (as we saw in chapter 6). Finally, we call the API's `Start()` method, which starts up the API server and listens for requests. Here is where we make our first change. Instead of calling `api.Start()` in our main goroutine, we call it using a third goroutine, which allows us to run all the necessary pieces of the worker concurrently.

Listing 7.10. We re-use the `main()` function from the previous chapter and make one minor change to run all the worker's components in separate goroutines.

```
func main() {
        host := os.Getenv("CUBE_HOST")
        port, _ := strconv.Atoi(os.Getenv("CUBE_PORT"))

        fmt.Println("Starting Cube worker")

        w := worker.Worker{
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]*task.Task),
        }
        api := worker.Api{Address: host, Port: port, Worker: &w}

        go runTasks(&w)
        go w.CollectStats()
        go api.Start()
```

At this point, we have an instance of our worker running. Now, we want to start an instance of our manager.

To do this, we create a list of workers and assign it to a variable named `workers`. This list is a slice of strings, and we add our single worker to it. Next, we create an instance of our manager by calling the `New` function created earlier, passing it our list of workers.

Listing 7.11. TODO

```
    workers := []string{fmt.Sprintf("%s:%d", host, port)}
```

```
        m := manager.New(workers)
```

With an instance of our worker running and an instance of a manager created, the next step is to add some tasks to the manager. In listing 7.11, we create three tasks. This is a random decision. Feel free to choose more or less. Creating the `Task` and `TaskEvent` should look familiar, since it's the same thing we've done in previous chapters in working with the worker. Now, however, instead of adding the `TaskEvent` to the worker directly, we add it to the manager by calling `AddTask` on the manager `m` and passing it the task event `te`. The final step in this loop is to call the `SendWork` method on manager `m`, which will select the only worker we currently have and, using the worker's API, send the worker the task event.

Listing 7.12. To exercise our newly implemented manager, we create three tasks and add them to the manager's queue before calling the manager's `SendWork` method.

```
for i := 0; i < 3; i++ {
                t := task.Task{
                        ID:    uuid.New(),
                        Name:  fmt.Sprintf("test-container-%d", i
                        State: task.Scheduled,
                        Image: "strm/helloworld-http",
                }
                te := task.TaskEvent{
                        ID:    uuid.New(),
                        State: task.Running,
                        Task:  t,
                }
                m.AddTask(te)
                m.SendWork()
        }
```

Reaching this point, let's pause for a moment and think about what has happened.

- We have created an instance of the worker and it's running and listening for API requests
- We have created an instance of the manager, and it has a list of workers containing the single worker we created earlier
- We have created three tasks and added those tasks to the manager
- The manager has selected a worker (in this case, the only one that exists)

and sent it the three tasks
- The worker has received the tasks and at least attempted to start them

From this list, it's clear there are two perspectives on the state of the tasks in this system. There is the manager's perspective, and there is the worker's perspective. From the manager's perspective, it has sent the tasks to the worker. Unless there is an error returned from the request to the worker's API, the manager's work in this process could be considered complete.

From the worker's perspective, things are more complicated. It's true the worker has received the request from the manager. Upon receiving the request, however, we must remember how we built the worker. The worker API's request handlers don't directly perform any operations; instead, the handlers take the requests and put them on the worker's queue. In a separate goroutine, the worker performs the direct operations to start and stop tasks. As we mentioned in chapter 5, this design decision allows us to separate the concern of handling API requests from the concern of performing the actual operations to start and stop tasks.

Once the worker picks a task off of its queue and attempts to perform the necessary operation, any number of things can go wrong. As we enumerated earlier, examples of things going wrong can include user errors (e.g. a user specifying a non-existing Docker image in the task specification) or machine errors (e.g. a machine doesn't have enough disk space to download the task's Docker image).

As we can see, in order for the manager to be an effective component in our orchestration system, it can't just use a fire-and-forget approach to task management. It must constantly check in with the workers it is managing to reconcile its perspective with that of the workers'.

We discussed this problem earlier in the chapter, and it was our motivation for implementing the manager's `UpdateTasks` method. So, now, let's make use of our foresight. Once the manager has sent tasks off to the worker, we want to call the manager's `UpdateTasks` method.

To accomplish our objective, we'll use another goroutine, and this goroutine will call an *anonymous function*. Like other programming languages, an

anonymous function in Go is simply a function that is defined where it's called. Inside of this anonymous function, we use an infinite loop. Inside this loop, we print an informative log message that tells us the manager is updating tasks from its workers. Then, we call the manager's `UpdateTasks` method to actually update its perspective on the tasks in the system. And, finally, it sleeps for fifteen seconds. As we've done previously, we're using sleep here purely for the purpose of slowing the system down so we can observe and understand our work.

While we're on the topic of observing our work, let's also add another infinite loop that ranges over the tasks and prints out the `ID` and `State` of each task in the system. This will allow us to observe the tasks' state changing as the `UpdateTasks` method does its job.

Listing 7.13. This pattern of using an anonymous function to run a piece of code in a separate goroutine is common in the Go ecosystem.

```
go func() { #1
        for { #2
                fmt.Printf("[Manager] Updating tasks from %d work
                m.UpdateTasks() #3
                time.Sleep(15 * time.Second)
        }
}() #4

for { #5
        for _, t := range m.TaskDb { #6
                fmt.Printf("[Manager] Task: id: %s, state: %d\n",
                time.Sleep(15 * time.Second)
        }
}
```

At this point, we've written all the necessary code to run our worker and manager together. So, let's switch from writing code to running it.

To run our `main.go` program, call it with `go run main.go`. Also, it's important to define the `CUBE_HOST` and `CUBE_PORT` environment variables as part of the command, as this tells the worker API on what port to listen. These environment variables will also be used by the manager to populate it's `Workers` field. When we start our program, the initial output should look familiar. We should see the following:

```
$ CUBE_HOST=localhost CUBE_PORT=5555 go run main.go
Starting Cube worker #1
2022/01/30 14:17:12 Pulled {9f122e79-6623-4986-a9df-38a5216286fb
2022/01/30 14:17:12 No tasks to process currently. #3
2022/01/30 14:17:12 Sleeping for 10 seconds.
2022/01/30 14:17:12 Collecting stats #4
2022/01/30 14:17:12 Added task 9f122e79-6623-4986-a9df-38a5216286
```

After the initial output above, we should see the worker start the tasks. Then, once all three tasks are started, you should start seeing output like that below from the `main.go` program. This is our code that is calling the manager's `UpdateTasks` method in one `for` loop, and ranging over the manager's tasks and printing out the `ID` and `State` of each task in a separate `for` loop.

```
[Manager] Updating tasks from 1 workers
[Manager] Task: id: 9f122e79-6623-4986-a9df-38a5216286fb, state:
[Manager] Updating tasks from 1 workers
[Manager] Task: id: 792427a7-e306-44ef-981a-c0b76bfaab8e, state:
```

Interleaved in the output, you should also see output like that below. This output is coming from our manager itself.

```
2022/01/30 14:18:57 Checking worker localhost:5555 for task updat
2022/01/30 14:18:57 Attempting to update task 792427a7-e306-44ef-
2022/01/30 14:18:57 Attempting to update task 2507e136-7eb7-4530-
2022/01/30 14:18:57 Attempting to update task 9f122e79-6623-4986-
```

While our `main.go` program is running in one terminal, open a second terminal and query the worker API. Depending on how quickly you run the `curl` command after starting up the `main.go` program, you may not see all three tasks. Eventually, though, you should see them.

```
$ curl http://localhost:5555/tasks |jq .
[
  {
    "ID": "723143b3-4cb8-44a7-8dad-df553c15bce3",
    "ContainerID": "14895e61db8d08ba5d0e4bb96d6bd75023349b53eb4ba
    "Name": "test-container-0",
    "State": 2,
    [....]
  },
  {
    "ID": "a85013fb-2918-47fb-82b0-f2e8d63f433b",
    "ContainerID": "f307d7045a36501059092f06ff3d323e6246a7c854bfa
```

```
    "Name": "test-container-1",
    "State": 2,
    [....]
  },
  {
    "ID": "7a7eb0ef-8516-4103-84a7-9f964ba47cb8",
    "ContainerID": "fffc1cf5b8ca7d33eb3c725f4190b81e0978f3efc8405
    "Name": "test-container-2",
    "State": 2,
    [....]
  }
]
```

In addition to querying the worker API, we can use the `docker` command to verify that our tasks are indeed running. Note that I've removed some of the columns from the output of `docker ps` for readability.

```
$ docker ps
CONTAINER ID   CREATED         STATUS         NAMES
fffc1cf5b8ca   5 minutes ago   Up 5 minutes   test-container-2
f307d7045a36   5 minutes ago   Up 5 minutes   test-container-1
14895e61db8d   5 minutes ago   Up 5 minutes   test-container-0
```

7.6 Summary

- The manager records user requests in the form of `task.TaskEvent` items and stores them in its `EventDB`. This task event, which includes the `task.Task` itself, serves as the user's desired state for the task.
- The manager records the *state of the world*, i.e. the actual state of a task from the perspective of a worker, in its `TaskDB`. For this initial implementation of the manager, we do not attempt to retry failed tasks and instead simply record the state. We will revisit this issue in a later chapter.
- The manager serves a purely administrative function. It accepts requests from users, records those requests in its internal databases, then selects a worker to run the task and passes it along to the worker. It periodically updates its internal state by querying the worker's API. It is not directly involved in any of the operations to actually run a task.
- We've used a simple, extremely naive algorithm to assign tasks to workers. This decision allowed us to code a working implementation of the manager in a relatively small number of lines of code. We will

revisit this decision in a later chapter.

8 An API for the manager

This chapter covers

- Understanding the purpose of the manager API
- Implementing methods to handle API requests
- Creating a server to listen for API requests
- Starting, stopping, and listing tasks via the API

In chapter 7, we implemented the core functionality of the manager component: pulling tasks off its queue, selecting a worker to run those tasks, sending them to the selected workers, and periodically updating the state of tasks. That functionality is just the foundation and doesn't provide a simple way for users to interact with the manager.

So, like we did with the worker in chapter 5, we're going to build an API for the manager. This API will wrap the manager's core functionality and expose it to users. In the case of the manager, `users` means *end users*, that is, developers who want to run their application in our orchestration system.

The manager's API, like the worker's, will be simple. It will provide the means for users to perform these basic operations:

- Send a task to the manager
- Get a list of tasks
- Stop a task

This API will be constructed using the same components that we used for the worker's API. It will be comprised of *handlers*, *routes*, and a *mux*.

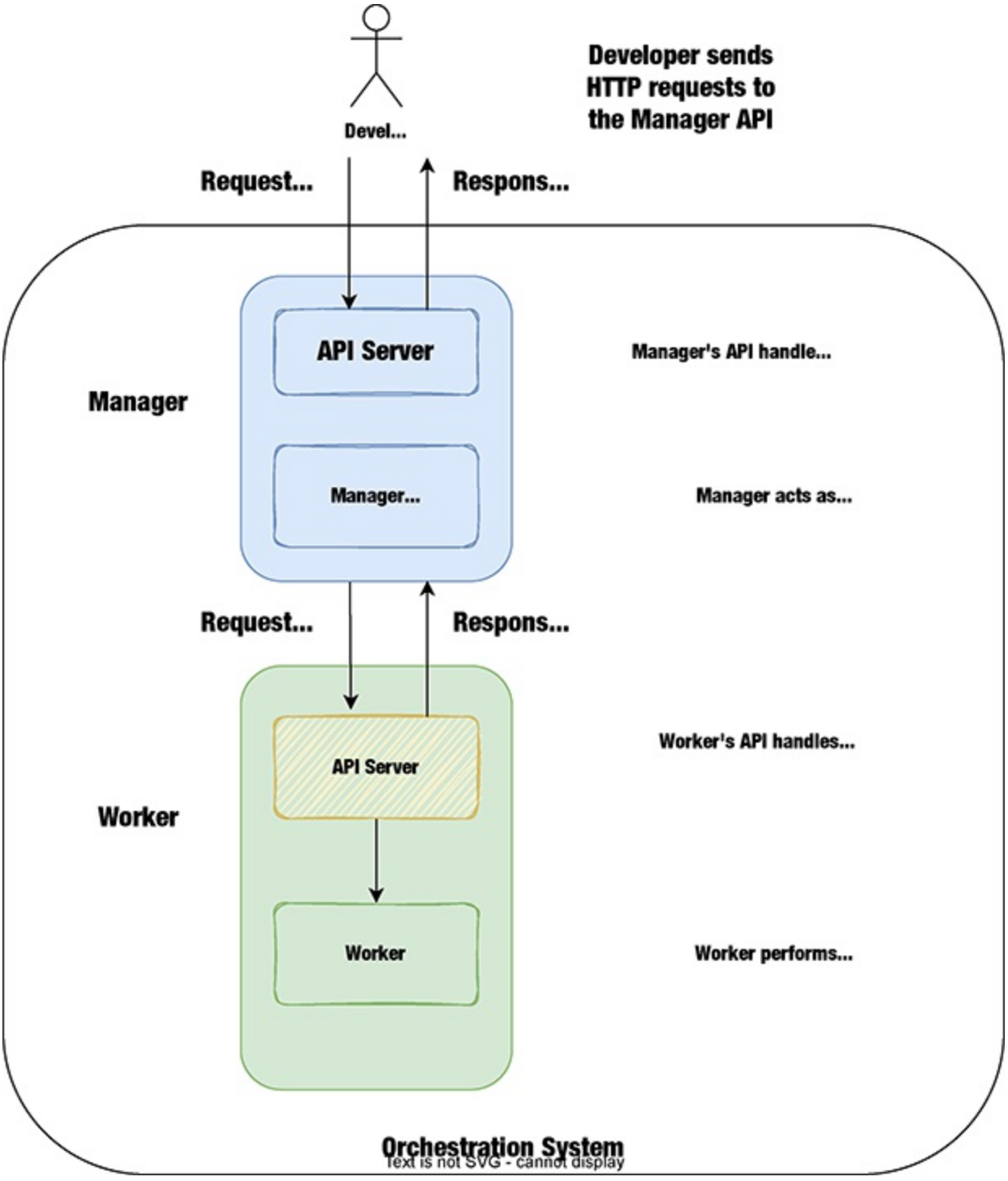
8.1 Overview of the manager API

Before we get too far into our code, let's zoom out for a minute and take a more holistic view of where we're going. We've been focusing pretty tightly for the last couple of chapters, so it will be a good reminder to see how the

technical details fit together.

We’re not building a manager and worker just for the sake of it. The purpose of building them is to fulfill a need: developers need a way to run their applications in a reliable and resilient way. The manager and worker are abstractions that free the developer from having to think too deeply about the underlying infrastructure (whether physical or virtual) on which their applications run. Figure 8.1 reminds us what this abstraction looks like.

Figure 8.1. The manager is comprised of an API server and Manager components, and similarly the worker is comprised of an API server and worker components. The user communicates with the manager, and the manager communicates with one or more workers.



With that reminder, let’s zoom back in to the details of constructing the manager’s API. Because it will be similar to the worker’s, we won’t spend as much time going into the details of handlers, routes, and muxes. If you need a refresher, please refer to section 5.1 in chapter 5.

8.2 Routes

Let's start by identify the routes that our manager API should handle. And, it shouldn't be too surprising that the routes are identical to that of the worker's API. In some ways, our manager is acting as a reverse proxy: instead of balancing requests for, say, web pages across a number of web servers, it's balancing requests to run tasks across a number of workers.

Thus, like the worker's, the manager's API will handle `GET` requests to `/tasks`, which will return a list of all the tasks in the system. This enables our users to see what tasks are currently in the system. It will handle `POST` requests to `/tasks`, which will start a task on a worker. This allows our users to run their tasks in the system. And, finally, it will handle `DELETE` requests to `/tasks/{taskID}`, which will stop a task specified by the `taskID` in the route. This allows our users to stop their tasks.

Table 8.1. Routes used by our manager API

Method	Route	Description
GET	/tasks	get a list of all tasks
POST	/tasks	create a task
DELETE	/tasks/{taskID}	stop the task identified by `taskID`

8.3 Data format, requests, and responses

If the routes we use for the manager's API are similar to the worker's, then what about the data format, and the requests and responses that the manager will receive and return? Again, it should not be surprising that the manager's API will use the same data format, JSON, as the worker's API. If the worker

API speaks JSON, the manager's API should speak the same language to minimize unnecessary translation between data formats. Thus, any data sent to the manager's API must be JSON-encoded, and any data returned by the API will also be encoded as JSON.

Table 8.2. An updated route table showing whether the routes send a request body, returns a response body, and the status code for a successful request.

Method	Route	Description	Request Body	Response Body	Status code
GET	/tasks	get a list of all tasks	none	list of tasks	200
POST	/tasks	create a task	JSON-encoded task.TaskEvent	none	201
DELETE	/tasks/{taskID}	stop the task identified by `taskID`	none	none	204

We can see how these routes are used in figure 8.2, which shows a POST request to create a new task and a GET request to get a list of tasks. The developer issues requests to the Manager, and the Manager returns responses. In the first example, the developer issues a POST request with a body that specifies a task to run. The manager responds with a status code of `201`. In the second example, the developer issues a GET request, and the manager responds with a status code of `200` and a list of its tasks.

Figure 8.2. Two examples of how a developer uses the Manager's API.

Request 1

http://localhost:5556/tasks
POST


```
{
  ID: 98dfdf49-1e65-42c9-b8bd-183805f44605,
  State: task.Running,
  Task: {
    ID: d6ce68e1-e4aa-4842-b3bd-8379283792d5,
    Name: "task-1",
    State: task.Scheduled,
    Image: "strm/helloworld-http",
  }
}
```


Response 1

HTTP/1.1 201 Created


```
{
  ID: 98dfdf49-1e65-42c9-b8bd-183805f44605,
  State: task.Running,
  Task: {
    ID: d6ce68e1-e4aa-4842-b3bd-8379283792d5,
    Name: "task-1",
    State: task.Scheduled,
    Image: "strm/helloworld-http",
  }
}
```


Request 2

http://localhost:5555/tasks
GET

Response 2

HTTP/1.1 200 OK


```
[
  {
    "ID":"98dfdf49-1e65-42c9-b8bd-183805f44605",
    "ContainerID":"6df4e15a5c840b",
    "Name":"test-chapter-5-1",
    "State":2,

    "Image":"strm/helloworld-http",
    "Memory":0,
    "Disk":0,

    "ExposedPorts":null,
    "PortBindings":null,
    "RestartPolicy":"",
    "StartTime":"0001-01-01T00:00:00Z",
    "FinishTime":"0001-01-01T00:00:00Z"
  }
]
```


Text is not SVG - cannot display

8.4 The API struct

Drilling down a little farther, we notice another similarity with the worker's API. The manager's API also uses an `Api` struct, which will encapsulate the necessary behavior of its API. The only difference will be swapping out a single field: the `Worker` field gets replaced by a `Manager` field, which will contain a pointer to an instance of our manager. Otherwise, the `Address`, `Port`, and `Router` fields all have the same types and serve the same purposes as they did in the worker API.

Listing 8.1. The manager's `Api` struct functions similarly to the worker's.

```
type Api struct {
        Address string
        Port    int
        Manager *Manager
        Router  *chi.Mux
}
```

8.5 Handling requests

Continuing to drill down, let's talk about the handlers for the manager API. These should look familiar, as they are the same three handlers we implemented for the worker:

- StartTaskHandler(w http.ResponseWriter, r *http.Request)
- GetTasksHandler(w http.ResponseWriter, r *http.Request)
- StopTaskHandler(w http.ResponseWriter, r *http.Request)

We'll implement these handlers in a file named `handlers.go`, which you should create in the `manager` directory next to the existing `manager.go` file. To reduce the amount of typing necessary, feel free to copy the handlers from the worker's API and paste them into the manger's `handlers.go` file. The only changes we'll need to make are to update any references to `a.Worker` to `a.Manager`.

Let's start with the `StartTaskHandler`, which works the same as its worker counterpart. It expects a request body encoded as JSON. It decodes that

request body into a `task.TaskEvent`, checking for any decoding errors. Then it adds the task event to the Manager's queue using the manager's `AddTask` method implemented in chapter 7. The implementation can be seen here in listing 8.2.

Listing 8.2. The manager's `StartTaskHandler`.

```
func (a *Api) StartTaskHandler(w http.ResponseWriter, r *http.Req
        d := json.NewDecoder(r.Body) #1
        d.DisallowUnknownFields()

        te := task.TaskEvent{}
        err := d.Decode(&te) #2
        if err != nil { #3
                msg := fmt.Sprintf("Error unmarshalling body: %v\
                log.Printf(msg)
                w.WriteHeader(400)
                e := ErrResponse{
                        HTTPStatusCode: 400,
                        Message:        msg,
                }
                json.NewEncoder(w).Encode(e)
                return
        }

        a.Manager.AddTask(te) #4
        log.Printf("Added task %v\n", te.Task.ID)
        w.WriteHeader(201) #5
        json.NewEncoder(w).Encode(te.Task) #6
}
```

Next up is the `GetTasksHandler`. Like the `StartTaskHandler` above, `GetTasksHandler` works the same way as its counterpart in the worker API. The only change is that this API gets the tasks from the manager by passing `a.Manager.GetTasks()` to the encoding method.

Listing 8.3. The manager's `GetTasksHandler`.

```
func (a *Api) GetTasksHandler(w http.ResponseWriter, r *http.Requ
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(200)
        json.NewEncoder(w).Encode(a.Manager.GetTasks())
}
```

Finally, let's implement the `StopTaskHandler`. Again, the method works the same way as its worker counterpart, so there isn't much new to add to the discussion.

Listing 8.4. The manager's `StopTaskHandler`.

```
func (a *Api) StopTaskHandler(w http.ResponseWriter, r *http.Requ
        taskID := chi.URLParam(r, "taskID") #1
        if taskID == "" {
                log.Printf("No taskID passed in request.\n")
                w.WriteHeader(400)
        }

        tID, _ := uuid.Parse(taskID)
        _, ok := a.Manager.TaskDb[tID] #2
        if !ok {
                log.Printf("No task with ID %v found", tID)
                w.WriteHeader(404)
        }

        te := task.TaskEvent{ #3
                ID:        uuid.New(),
                State:     task.Completed,
                Timestamp: time.Now(),
        }
        taskToStop := a.Manager.TaskDb[tID] #4
        taskCopy := *taskToStop #5
        taskCopy.State = task.Completed
        te.Task = taskCopy
        a.Manager.AddTask(te) #6

        log.Printf("Added task event %v to stop task %v\n", te.ID
        w.WriteHeader(204) #7
}
```

As a reminder, as we did in the Worker, the Manager's handler methods are not operating directly on tasks. We have separated the concerns of responding to API requests and the operations to start and stop tasks. So the API simply puts the request on the Manager's queue via the `AddTask` method, then the manager picks up the task from its queue and performs the necessary operation.

So, we've been able to implement the handlers in the manager's API by

copying and pasting the handlers from the worker's API and making a few minor adjustments. At this point, we've implemented the meat of the API.

8.6 Serving the API

Now that we have implemented the manager's handlers, let's complete our work that will let us serve the API to users. We'll start by copying and pasting the `initRouter` method from the worker. This method sets up our router and creates the required endpoints, and since the endpoints will be the same as the worker's, we don't need to modify anything.

Listing 8.5. The manager API's `initRouter` method defines the routes that will be served to users.

```
func (a *Api) initRouter() {
        a.Router = chi.NewRouter()
        a.Router.Route("/tasks", func(r chi.Router) {
                r.Post("/", a.StartTaskHandler)
                r.Get("/", a.GetTasksHandler)
                r.Route("/{taskID}", func(r chi.Router) {
                        r.Delete("/", a.StopTaskHandler)
                })
        })
}
```

For the icing on the cake, let's take care of starting the API by copying the `Start` method from the worker's API. The manager's API will start up in the same way, so like the `initRouter` method above, we don't need to make any changes.

Listing 8.6. The manager API's `Start` method starts the API server, enabling it to respond to user requests.

```
func (a *Api) Start() {
        a.initRouter()
        http.ListenAndServe(fmt.Sprintf("%s:%d", a.Address, a.Por
}
```

8.7 A few refactorings to make our lives easier

At this point, we've implemented everything we need to have a functional

API for the manager. But, now that we're to a point in our journey where we have APIs for both the worker and the manager, let's make a few minor tweaks that will make it easier to run these APIs.

If you recall chapter 5, we created the function `runTasks` in our `main.go` file. We used it as way to continuously check for new tasks the worker needed to run. If it found any tasks in the worker's queue, then it called the workers `RunTask` method.

Instead of having this function be part of the `main.go` file, let's move it into the worker itself. This change will then encapsulate all the necessary worker behavior in the `Worker` object. So copy the `runTasks` function from the `main.go` file and paste it into the `worker.go` file. Then, to clean everything up so the code will run we're going to make three changes:

- Rename the existing `RunTask` method to `runTask`
- Rename the `runTasks` method from `main.go` to `RunTasks`
- Change the `RunTasks` method to call the newly renamed `runTask` method

You can see these changes below in listing 8.7.

Listing 8.7. Moving the `runTasks` function from main.go to the worker and renaming it `RunTasks`.

```
func (w *Worker) runTask() task.DockerResult { #1
    // body of the method stays the same
}

func (w *Worker) RunTasks() { #2
        for {
                if w.Queue.Len() != 0 {
                        result := w.runTask() #3
                        if result.Error != nil {
                                log.Printf("Error running task: %
                        }
                } else {
                        log.Printf("No tasks to process currently
                }
                log.Println("Sleeping for 10 seconds.")
                time.Sleep(10 * time.Second)
        }
```

```
}
```

In a similar fashion, we're going to make some changes to the `Manager` struct that will make it easier to use the manager in our `main.go` file. For starters, let's rename the manager's `UpdateTasks` method to `updateTasks`. So the method should look like this:

```
func (m *Manager) updateTasks() {
    // body of the method stays the same
}
```

Next, let's create a new method on our `Manager` struct called `UpdateTasks`. This method serves a similar purpose to the `RunTasks` method we added to the worker. It runs an endless loop, inside which it calls the manager's `updateTasks` method. This change makes it possible for us to remove the anonymous function we used in chapter 7's `main.go` file that performed the same function.

Listing 8.8. We add the `UpdateTasks` method to the manager, which performs a similar role the worker's `RunTasks` method.

```
func (m *Manager) UpdateTasks() {
        for {
                log.Println("Checking for task updates from worke
                m.updateTasks()
                log.Println("Task updates completed")
                log.Println("Sleeping for 15 seconds")
                time.Sleep(15 * time.Second)
        }
}
```

Finally, let's add the `ProcessTasks` method you see in listing 8.9 below to the Manager. This method also works similar to the worker's `RunTasks` method: it runs an endless loop, repeatedly calling the manager's `SendWork` method.

Listing 8.9. The manager's `ProcessTasks` method.

```
func (m *Manager) ProcessTasks() {
        for {
                log.Println("Processing any tasks in the queue")
                m.SendWork()
```

```
                log.Println("Sleeping for 10 seconds")
                time.Sleep(10 * time.Second)
        }
}
```

8.8 Putting it all together

Alright, it's that's time again: time to take what we've built in the chapter and run it. Before we do that, however, let's quickly summarize what it is that we've built so far:

- We have wrapped the manager component in an API that allows users to communicate with the manager
- We've constructed the manager's API using the same types of components we used for the worker's API: *handlers*, *routes*, and a *router*
- The *router* listens for requests to the *routes*, and dispatches those requests to the appropriate *handlers*

So, let's start by copying and pasting the `main` function from the `main.go` file in chapter 7. This will be our starting point.

There is one major difference between our situation at the end of this chapter and that of the last. That is, we now have APIs for both the worker and the manager. So, while we will be creating instances of each, we will not be interacting with them directly as we have in the past. Instead, we will be passing these instances into their respective APIs, then starting those APIs so they are listening to HTTP requests.

In past chapters where we have started instances of the worker's API, we have set two environment variables: `CUBE_HOST` and `CUBE_PORT`. These were used to set up the worker API to listen for requests at `http://localhost:5555`. Now, however, we have two APIs that we need to start. To handle our new circumstances, let's set up our main function to extract a set of `host:port` environment variables for each API. As you can see in listing 8.10, these will be called `CUBE_WORKER_HOST`, `CUBE_WORKER_PORT`, `CUBE_MANAGER_HOST`, and `CUBE_MANAGER_PORT`.

Listing 8.10. Extracting the host and port for each API from environment variables.

```
func main() {
        whost := os.Getenv("CUBE_WORKER_HOST")
        wport, _ := strconv.Atoi(os.Getenv("CUBE_WORKER_PORT"))

        mhost := os.Getenv("CUBE_MANAGER_HOST")
        mport, _ := strconv.Atoi(os.Getenv("CUBE_MANAGER_PORT"))
}
```

Next, after extracting the host and port values from the environment and storing them in appropriately named variables, let's start up the worker API. This process should look familiar from chapter 7. The only difference here, however, is that we're calling the `RunTasks` method on the worker object, instead of a separate `runTasks` function that we previously defined in `main.go`. As we did in chapter 7, we call each of these methods using the `go` keyword, thus running each in a separate goroutine.

Listing 8.11. Starting up the worker API.

```
    fmt.Println("Starting Cube worker")

        w := worker.Worker{
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]*task.Task),
        }
        wapi := worker.Api{Address: whost, Port: wport, Worker: &

        go w.RunTasks()
        go w.CollectStats()
        go wapi.Start()
```

Finally, we'll start up the manager API. This process starts out the same as it did in the last chapter. We create a list of workers that contains the single worker created above, represented as a string by its `<host>:<port>`. Then, we create an instance of our manager, passing in the list of `workers`. Next, we create an instance of the manager's API and store it in a variable called `mapi`.

The next two lines in our `main` function set up two goroutines that will run in parallel with the main goroutine running the API. The first goroutine will run the manager's `ProcessTasks` method. This ensures the manager will process any incoming tasks from users. The second goroutine will run the manager's

`UpdateTasks` method. It will ensure the manager updates the state of tasks by querying the worker's API to get up-to-date states for each task.

Then, what we've been waiting for. We start the manager's API by calling the `Start` method.

Listing 8.12. Staring up the manager API.

```
    fmt.Println("Starting Cube manager")

        workers := []string{fmt.Sprintf("%s:%d", whost, wport)}
        m := manager.New(workers)
        mapi := manager.Api{Address: mhost, Port: mport, Manager:

        go m.ProcessTasks()
        go m.UpdateTasks()

        mapi.Start()

}
```

At this point, all that's left to do is to run our main program so both the manger and worker APIs are running. As you can see below, both the worker and the manager get started.

```
$ CUBE_WORKER_HOST=localhost \
CUBE_WORKER_PORT=5555 \
CUBE_MANAGER_HOST=localhost \
CUBE_MANAGER_PORT=5556 \
go run main.go
Starting Cube worker
Starting Cube manager
2022/03/06 14:45:47 Collecting stats
2022/03/06 14:45:47 Checking for task updates from workers
2022/03/06 14:45:47 Processing any tasks in the queue
2022/03/06 14:45:47 Checking worker localhost:5555 for task updat
2022/03/06 14:45:47 No work in the queue
2022/03/06 14:45:47 Sleeping for 10 seconds
2022/03/06 14:45:47 No tasks to process currently.
2022/03/06 14:45:47 Sleeping for 10 seconds.
2022/03/06 14:45:47 Task updates completed
2022/03/06 14:45:47 Sleeping for 15 seconds
```

Let's do a quick sanity check to verify our manager does indeed respond to

requests. Let's issue a `GET` requests to get a list of the tasks it knows about. It should return an empty list.

```
$ curl -v localhost:5556/tasks
*   Trying 127.0.0.1:5556...
* Connected to localhost (127.0.0.1) port 5556 (#0)
> GET /tasks HTTP/1.1
> Host: localhost:5556
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Sun, 06 Mar 2022 19:52:46 GMT
< Content-Length: 3
<
[]
```

Cool! Our manager is listening for requests as we expected. As we can see, though, it doesn't have any tasks—because we haven't told it to run any, yet. So, let's take the next step and send a request to the manager that instructs it to start a task for us.

For this purpose, let's create a file named `task.json` in the same directory where our `main.go` file is. Inside this file, let's create the JSON-representation of a task as seen below in listing 8.13. This representation similar to what we used in `main.go` in chapter 7, except we're moving out into a separate file.

```
{
    "ID": "6be4cb6b-61d1-40cb-bc7b-9cacefefa60c",
    "State": 2,
    "Task": {
        "State": 1,
        "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
        "Name": "test-chapter-5",
        "Image": "strm/helloworld-http"
    }
}
```

Now that we've created our `task.json` file with the task we want to send to the manager via it's API, let's use `curl` to send a POST request to the

manager API's `/tasks` endpoint. And, as expected, the manger's API responds with a `201` response code.

```
$ curl -v --request POST \
--header 'Content-Type: application/json' \
--data @task.json \
localhost:5556/tasks
*   Trying 127.0.0.1:5556...
* Connected to localhost (127.0.0.1) port 5556 (#0)
> POST /tasks HTTP/1.1
> Host: localhost:5556
> User-Agent: curl/7.81.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 230
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 201 Created
< Date: Sun, 06 Mar 2022 20:04:36 GMT
< Content-Length: 286
< Content-Type: text/plain; charset=utf-8
<
{
    "ID":"21b23589-5d2d-4731-b5c9-a97e9832d021",
    "ContainerID":"",
    "Name":"test-chapter-5",
    "State":1,
    "Image":"strm/helloworld-http",
    "Cpu":0,
    "Memory":0,
    "Disk":0,
    "ExposedPorts":null,
    "PortBindings":null,
    "RestartPolicy":"",
    "StartTime":"0001-01-01T00:00:00Z",
    "FinishTime":"0001-01-01T00:00:00Z"
}
```

One thing to note about the JSON return by the manager's API. Notice that the `ContainerID` field is empty. The reason for this is that, like the worker's API, the manager's API doesn't operate directly on tasks. As tasks come in to the API, they are added to the manager's queue, and the manager works on them independently of the request. So at the time of our request, the manager hasn't shipped of the task to the worker, and so it can't know what the `ContainerID` will be. If we make a subsequent request to the manager's API

to `GET /tasks`, we should see a `ContainerID` for our task.

```
$ curl -v localhost:5556/tasks|jq
*   Trying 127.0.0.1:5556...
> GET /tasks HTTP/1.1
> Host: localhost:5556
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Sun, 06 Mar 2022 20:16:43 GMT
< Content-Length: 352
[
  {
    "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
    "ContainerID": "428115c14a41243ec29e5b81feaccbf4b9632e2caaeb5
    "Name": "test-chapter-8",
    "State": 2,
    "Image": "strm/helloworld-http",
    "Cpu": 0,
    "Memory": 0,
    "Disk": 0,
    "ExposedPorts": null,
    "PortBindings": null,
    "RestartPolicy": "",
    "StartTime": "0001-01-01T00:00:00Z",
    "FinishTime": "0001-01-01T00:00:00Z"
  }
]
```

There is one minor thing to keep in mind when querying the manager's API, as we did above. Depending on how quickly we issue our `GET /tasks` request after sending the initial `POST /tasks` request, we may still not see a `ContainerID`. Why is that? If you recall, the manager updates its view of tasks by making a `GET /tasks` requests to the worker's API. It then uses the response to that request to update the state of the tasks in its own datastore. If you look back to figure 8.12 above, you can see that our `main.go` program is running the manager's `UpdateTasks` method in a separate goroutine, and that method sleeps for 15 seconds between each attempt to update tasks.

Once the manager shows the task running—i.e., we get a `ContainerID` in our `GET /tasks` response—we can further verify the task is indeed running using

the `docker ps` command.

```
$ docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}\t{{.
CONTAINER ID   IMAGE                  STATUS              NAMES
428115c14a41   strm/helloworld-http   Up About a minute   test-ch
```

So, now that we've seen that we can use the manager's API to start a task and to get a list of tasks, let's use it to stop our running task. To do this, we issue a `DELETE /tasks/{taskID}` request, like that below.

```
$ curl -v --request DELETE \
    'localhost:5556/tasks/21b23589-5d2d-4731-b5c9-a97e9832d021'
*   Trying 127.0.0.1:5556...
* Connected to localhost (127.0.0.1) port 5556 (#0)
> DELETE /tasks/21b23589-5d2d-4731-b5c9-a97e9832d021 HTTP/1.1
> Host: localhost:5556
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 204 No Content
< Date: Sun, 06 Mar 2022 20:29:27 GMT
```

As we can see, the manager's API accepted our request, and it responded with a `204` response, as expected. You should also see in the following output from our `main.go` program. (Note I've truncated some of the output to make it easier to read.)

```
2022/03/06 15:29:27 Added task event 937e85eb to stop task 21b235
Found task in queue:
2022/03/06 15:29:32 attempting to transition from 2 to 3
2022/03/06 15:29:32 Attempting to stop container 442a439de
2022/03/06 15:29:43 Stopped and removed container 442a439de for t
```

Again, we can use the `docker ps` command to confirm that our manager did what we expected it to do, which in this case is to stop the task.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS
```

8.9 Summary

- Like the worker's API, the manager's wraps its core functionality and exposes it as a HTTP server. Unlike the worker's API, who's primary user is the manager, the primary user of the manager's API is end users, in other words developers. Thus, the manager's API is what users interact with in order to run their tasks on the orchestration system.
- The manager and worker APIs provide an abstraction over our infrastructure, either physical or virtual, that remove the need for developers to concern themselves with such low-level details. Instead of thinking about how their application runs on a machine, they only have to be concerned about how their application run in a container. If it runs as expected in a container on their machine, then it can run on any machine that's also running the same container framework (i.e. Docker).
- Like the worker API, the manager's is a simple REST-based API. It defines routes that enable users to create, query, and stop tasks. Also, when being sent data it expects that data to be encoded as JSON, and it likewise encodes any data it sends as JSON.

9 What could possibly go wrong?

This chapter covers

- Enumerating potential failures
- Exploring options for recovering from failures
- Implementing task health checks to recover from task crashes

At the beginning of chapter 4, in which we started the process of implementing our worker, we talked about the scenario of running a web server that serves static pages. In that scenario, we considered how to deal with the problem of our site gaining in popularity and thus needing to be resilient to failures in order to ensure we could serve our growing user base. The solution, we said, was to run multiple instances of our web server. In other words, we decided to scale horizontally, a common pattern for scaling. By scaling the number of web servers, we can ensure that a failure in any given instance of the web server does not bring our site completely down and, thus, unavailable to our users.

In this chapter, we're going to modify the above scenario slightly. Instead of serving static web pages, we're going to serve an API. This API is dead simple: it takes a POST request with a body, and it returns a response with the same body. In other words, it simply echoes the request in the response.

With that minor change to our scenario, this chapter will reflect on what we've built thus far and discuss a number of failure scenarios, both with our orchestrator and the tasks running on it. And, then, we will implement several mechanisms for handling a subset of failure scenarios.

9.1 Overview of our new scenario

So, our new scenario involves an API that takes the body of a request, and simply returns that body in the response to the user. The format of the body is JSON, and our API defines this format in the `Message` struct you see below.

```
type Message struct {
    Msg string
}
```

Thus, making a curl request to this API looks like this:

```
$ curl -X POST localhost:7777/ -d '{"Msg":"Hello, world!"}'
{"Msg":"Hello, world!"}
```

And, as you can see in the response, we do indeed get back the same body that we sent. In order to make this scenario simpler to use, I've gone ahead and built a Docker image that we can simply re-use throughout the rest of the chapter. So, to run that API locally, all you have to do is this:

```
$ docker run -it --rm --name echo timboring/echo-server:latest
```

(By the way, if you're interested in the source code for this API, you can find it in the `echo` directory in the downloadable source code for this chapter.)

Now, let's move on to talk about the number of ways this API can fail when running it as a task in our orchestration system.

9.2 Failure scenarios

Failures happen. ALL THE TIME! As engineers, we should expect this. Failures are the norm, not the exception. More importantly, failures happen at multiple levels:

- failures at the level of the application
- failures at the level of individual tasks
- failures at the level of the orchestration system

Let's walk through what failures at each of these levels might look like.

9.2.1 Application startup failure

A task can also fail to start because the task's application has a bug in its startup routine. For example, we might decide that we want to store each request our echo service receives, and to do that we add a database as a

dependency. Now, when an instance of our echo service starts up, it attempts to make a connection to the database.

What happens, however, if we can't connect to the database? Maybe it's down due to some networking issue. Or, if we're using a managed service, perhaps the administrators decided they needed to do some maintenance, and as a result the database service is unavailable for some period of time. Or, maybe the database is upset with its human overlords and has decided to go on strike.

It doesn't really matter why the database is unavailable. The fact is our application depends on the database, and it needs to do something when that database is not available. There are generally two options for how our application can respond:

- it can simply crash
- it can attempt to retry connecting to the database

In my experience, I've seen the former option used frequently. It's the default. As an engineer, I have an application, it needs a database, and I attempt to connect to the database when my application starts. Maybe I check for errors, log them, and then exit gracefully.

The latter option might be the better choice, but it adds some complexity. Today, most languages have at least one third-party library that provides the framework to perform retries using exponential backoff. Using this option, I could have my application attempt to connect to the database in a separate goroutine, and until it can connect maybe my application serves a 503 response with a helpful message explaining that the application is in the process of starting up.

9.2.2 Application bugs

A task can also fail after having successfully started up. For example, our echo service can start up and operate successfully for a while. But, we've recently added a new feature, and we didn't really test it thoroughly because we decided it was an important feature and getting it into production would allow us to post about it on Hacker News. A user queries our service in a way

that triggers our new code in an unexpected way and crashes the API. Ooops!

9.2.3 Task startup failures due to resource issues

A task can fail to start because the worker machine doesn't have enough resources (i.e. memory, cpu, or disk). In theory, this shouldn't happen. Orchestration systems like Kubernetes and Nomad implement sophisticated schedulers that take memory and cpu requirements into account when scheduling tasks onto worker nodes.

The story for disk, however, is a little more nuanced. Container images consume disk space. Run the `docker images` command on your machine and notice the `SIZE` column in the output. On my machine, the `timboring/echo-server` we're using in this chapter is 12.3MB in size. If if I pull down the `postgres:14` image, I can see that it's size is 376MB.

```
$ docker images
REPOSITORY              TAG           IMAGE ID       CREATED
timboring/echo-server   latest        fe039d2a9875   4 months ago
postgres                14            dd21862d2f49   4 days ago
```

While there are strategies to minimize size, images still reside on disk and thus consume space. In addition to container images, the other processes running on worker nodes also use disk space, for example they may store their logs on disk. Also, other containers running on a node may also be using disk space for their own data. So, it is possible that an orchestrator could schedule a task onto a worker node, and then when that worker starts up the task, the task fails because there isn't enough disk space to download the container image.

9.2.4 Task failures due to Docker daemon crashes and restarts

Running tasks can also be impacted by issues with the Docker daemon. For example, if the Docker daemon crashes, then our task will be terminated. Similarly, if we stop or restart the Docker daemon, then the daemon will also stop our task. This behavior is the default for the Docker daemon. Containers can be kept alive while the daemon is down using a feature called live restore, but the usage of that feature is beyond the scope of this book. For our

purposes, we will work with the default behavior.

9.2.5 Task failures due to machine crashes and restarts

The most extreme failure scenario is a worker machine crashing. In the case of an orchestration system, if a worker machine crashes, then the Docker daemon will obviously stop running, along with any tasks running on the machine.

Less extreme is the situation where a machine is restarted. Perhaps an administrator restarts it after updating some software, in which case the Docker daemon will be stopped and started back up after the machine reboots. In the process, however, any tasks will be terminated and will need to be restarted.

9.2.6 Worker failures

In addition to application and task failures, when we run the echo service on our orchestration system, the worker where the task runs can also fail. But, there is a little more involved at this level. When we say "worker", we need to clarify what exactly we're talking about. First, there is the worker component that we've written. And, second, there is the machine where our worker component runs.

So when we talk about worker failures, we actually have two discrete types of failures at this layer of our orchestration system. Our worker component that we've written can crash due to bugs in our code. It can also crash because the machine on which it's running crashes or becomes otherwise unavailable.

We already touched on machine failure above, but let's talk briefly about failures with the worker component. If it fails for some reason, what happens to the running tasks? Unlike the Docker daemon, our worker going down or restarting does not terminate running containers. It does mean that the manager cannot send *new* tasks to the worker, and it means that the manager cannot query the worker to get the current state of running tasks.

So, while a failure in our worker is inconvenient, it doesn't have an immediate impact on running tasks. (It could be more than inconvenient if, say, a number of workers crashed that resulted in your team not being able to deploy a mission critical bug fix. But, that's a topic for another book.)

9.2.7 Manager failures

The final failure scenario to consider involves the manager component. Remember, we said the manager serves an administrative function. It receives requests from users to run their tasks, and it schedules those tasks onto workers. Unless the manager and worker components are running on the same machine (and we wouldn't do that in a production environment, would we!), any issues with the manager will only effect those administrative functions.

So, if the manager component itself were to crash, or if the machine where it was running were to crash, then the impact would likely be minimal. Running tasks would continue to run. Users would not, however, be able to submit new tasks to the system because the manager would not be available to receive and take action on those requests. Again, inconvenient but not necessarily the end of the world.

9.3 Recovery options

As failures in an orchestration system can occur at multiple levels and have various degrees of impact, so too do the recovery options.

9.3.1 Recovery from application failures

As we mentioned above, applications can fail at startup due to external dependencies being unavailable. The only real automated recovery option here is to perform retries with exponential backoff or some other mechanism. An orchestration system cannot wave a magic wand and fix problems with external dependencies (unless, of course, that external dependency is also running on orchestration system). An orchestrator provides us with some tools for automated recovery from these kinds of situations, but if a database

is down, continually restarting the application isn't going to change the situation.

Similarly, an orchestration system can't help us with the bugs we introduce into our applications. The real solution on this front is tools like automated testing, which can help identify bugs before they are deployed into production.

9.3.2 Recovering from environmental failures

An orchestration system provides a number of tools for dealing with non-application specific failures. We can group the remaining failure scenarios together and call them *environmental* failures:

- failures with Docker
- failures with machines
- failures with an orchestrator's worker
- failures wiht an orchestrator's manager

Let's cover some ways in which an orchestration system can help recovering from these types of failures.

Recovering from task-level failures

Docker has a built-in mechanism for restarting containers when they exit. This mechanism is called *restart policies* and can be specified on the commandline using the `--restart` flag. In the example commandline below, we run a container using the `timboring/echo-server` image and tell Docker we want it to restart the container once if it exits because of a failure.

```
$ docker run \
    --restart=on-failure:1 \
    --name echo \
    -p 7777:7777 \
    -it \
    timboring/echo-server:bad-exit
```

Docker supports four restart policies:

- `no`: do nothing when a container exits (this is the default)
- `on-failure`: restart the container if it exits with a non-zero status code
- `always`: always restart the container, regardless of the exit code
- `unless-stopped`: always restart the container, except if the container was stopped

You can read more about Docker's restart policies in the docs at https://docs.docker.com/engine/reference/run/#restart-policies---restart.

The restart policy works well when dealing with individual containers being run outside of an orchestration system. In most production situations, we run Docker itself as a systemd unit. Systemd, as the initialization system for most Linux distributions, can ensure that applications that are supposed to be running are indeed actually running, especially after a reboot.

For containers running as part of an orchestration system, however, using Docker's restart policies can pose problems. The main issue is that it muddies the waters around who is responsible for dealing with failures. Is the Docker daemon ultimately responsible? Or is the orchestration system? Moreover, if the Docker daemon is involved in handling failures, then it adds complexity to the orchestrator because it will need to check with the Docker daemon to see if it's in the process of restarting a container.

For Cube, we will handle task failures ourselves instead of relying on Docker's restart policy. This decision, however, does raise another question. Should the manager or worker be responsible for handling failures?

The worker is closest to the task, so it seems natural to have the worker deal with failures. But, the worker is only aware of its own singular existence. Thus, it can only attempt to deal with failures in its own context. If it's overloaded, it can't make the decision to send the task to another worker, because it doesn't know about any other workers.

The manager, though farther away from the actual mechanisms that control task execution, has a broader view of the whole system. It knows about all the workers in the cluster, and it knows about the individual tasks running on each of those workers. Thus, the manager has more options for recovering from failures than does an individual worker. It can ask the worker running

the failed task to try to restart it. Or, if that worker is overloaded or unavailable (maybe it crashed), it can find another worker that has capacity to run the task.

9.3.3 Recovering from worker failures

As we mentioned above when discussing the types of worker failures, we said there are two distinct types of failures when it comes to the worker. There are failures in the worker component itself, and failures with the machine where the worker is running.

In the first case, when our worker component fails, we have the most flexibility. The worker itself isn't critical to existing tasks. Once a task is up and running, the worker is not involved in the task's ongoing operation. So, if the worker fails, there isn't much consequence to running tasks. In such a state, however, the worker is in a degraded state. The manager won't be able to communicate with the worker, which means it won't be able to collect task state, and it won't be able to place new tasks on the worker. It also won't be able to stop running tasks.

In this situation, we could have the manager attempt to fix the worker component. How? The obvious thing that comes to mind is for the manager to consider the worker dead and move all the tasks to another worker. This is a rather blunt force tactic, however, and could wreak more havoc. If the manager simply considers those tasks dead and attempts to restart them on another worker machine, what happens if the tasks are still running on the machine where the worker crashed? By blindly considering the worker and all of its tasks *dead*, the manager could be putting applications into an unexpected state. This is particularly true when there is only a single instance of a task.

The second case, where a worker machine has crashed, is also tricky. How are we defining "crashed"? Does it mean the manager cannot communicate with the worker via its API? Does it mean the manager performs some other operation to verify a worker is up, for example by attempting an ICMP ping? Moreover, can the manager be certain that a worker machine was actually down even if it did attempt an ICMP ping and did not receive a response?

What if the issue was that the worker machine's network card died, but the machine was otherwise up and operating normally? Similarly, what if the manager and worker machine were on different network segments, and a router, switch, or other piece of network equipment died, thus segmenting the two networks so that the manager could not talk to the worker machine?

As we can see, trying to make our orchestration system resilient to failures in the worker component is more complex than it may initially seem. It's difficult to determine if a machine is down—meaning it has crashed or been powered off and is otherwise not running any tasks—in which case the Docker daemon is not running, nor are any of the tasks under its control.

9.3.4 Recovering from manager failures

Finally, let's consider our options for failures in the manager component. Like the worker, there are two failure scenarios. The manager component itself could fail, and the machine on which the manager component is running could fail. While these scenarios are the same as in the worker, their impact and how we deal with them is slightly different.

First, if the manager dies—regardless of whether it's the manager component itself or the machine where it's running—there is no impact to running tasks. The tasks and the worker operate independently of the manager. In our orchestration system, if the manager dies, the worker and its tasks continue operating normally. The only difference is that the worker won't receive any new tasks.

Second, recovering from manager failures in our orchestration system will likely be less complex than recovering from failures at the worker layer. Remember, for the sake of simplicity, we have said that we will run only a single manager. So, if it fails, we only need to try to recover a single instance. If the manager component crashes, we can restart it. (Ideally, we'd run it using an init system like Systemd or supervisord.) If its datastore gets corrupted, we can restore it from a backup.

While not ideal, a manager failure doesn't bring our whole system down. It does cause some pain for developers, because while the manager is down

they won’t be able to start new tasks or stop existing ones. So, deployments of new features or bug fixes will be delayed until the manager is back online.

Obviously, the ideal state in regards to the manager would be to run multiple instances of the manager. This is what orchestrators like Borg, Kubernetes, and Nomad do. Like running multiple workers, multiple instances of the manager adds resiliency to the system as a whole. There is, however, added complexity.

When running multiple managers, we have to think about synchronizing state across all the instances. There might be a "primary" instance that is responsible for handling user requests and acting on them. This instance will also be responsible for distributing the state of all the system’s task across the other managers. If the primary instance fails, then another can take over its role. At this point, we start getting into the realm of consensus and the idea of how do systems agree on the state of the world. This is where things like the Raft protocol come into play, but going farther down this road is beyond the scope of this book.

9.4 Implementing health checks

With this complexity in mind, we are going to implement a simple solution for illustration purposes. We are going to implement health checks at the task level. The basic idea here is two-fold:

- An application implements a health check and exposes it on its API as `/health`. (The name of the endpoint could be anything, as long as its well defined and doesn’t change.)
- When a user submits a task, they define the health check endpoint as part of the task configuration.
- The manager calls a task’s health check periodically and will attempt to start a new task for any non-200 response.

With this solution, we don’t have to worry about whether the worker machine is reachable or not. We also don’t have to figure out if a worker component is working or not. We just call the health check for a task, and if it responds that it’s operating as expected, we know the manager can continue about its

business.

(Operationally, we still care about whether the worker component is functioning as expected. But, we can treat that issue separately from task health and how and when we need to attempt to restart tasks.)

There are two components to health checks. First, the worker has to periodically check the state of its tasks and update them accordingly. To do this, it can call the `ContainerInspect()` method on the Docker API. If the task is in any state other than `running`, then the worker updates the task's state to `Failed`.

Second, the manager must periodically call a task's health check. If the check doesn't pass (i.e. it returns anything other than a 200 response code), it then sends a task event to the appropriate worker to restart the task.

9.4.1 Inspecting a task on the worker

Let's start with refactoring our worker so it can inspect the state of a task's Docker container. If we remember back to chapter 3, we implemented the `Docker` struct seen below. The purpose of this struct is to hold a reference to an instance of the Docker client, which is what allows us to call the Docker API and perform various container operations. It also holds the `Config` for a task.

```
type Docker struct {
        Client *client.Client
        Config Config
}
```

In order to handle responses from the `ContainerInspect` API call, let's create a new struct called `DockerInspectResponse` in the `task/task.go` file. As we can see in listing 9.1, this struct will contain two fields. The `Error` field will hold an error if we encounter one when calling `ContainerInspect`. The `Container` field is a pointer to a `types.ContainerJSON` struct. This struct is defined in Docker's Go SDK (http://mng.bz/61a6). It contains all kinds of detailed information about a container, but most importantly for our purposes it contains the field `State`. This is the current state of the container as Docker

sees it.

Listing 9.1. The new `DockerInspectResponse` struct.

```
type DockerInspectResponse struct {
        Error     error
        Container *types.ContainerJSON
}
```

 **Note**

The concept of Docker container state can be confusing. For example, the doc (https://docs.docker.com/engine/reference/commandline/ps/#status) for the `docker ps` command mentions filtering by container *status*, where *status* is one of `created`, `restarting`, `running`, `removing`, `paused`, `exited`, or `dead`.

If you look at the source code, however, you'll find there is a `State` struct defined in `container/state.go` (http://mng.bz/oJdv) which looks like this:

```
type State struct {
    Running           bool
        Paused            bool
        Restarting        bool
        OOMKilled         bool
        RemovalInProgress bool
        Dead              bool
    // other fields omitted
}
```

As we can see, technically there is not a state called `created`, nor is there an `exited` state. So what is going on here? It turns out there is a method on the `State` struct named `StateString` (http://mng.bz/nJB4), and this is performing some logic that results in the statuses we see in the documentation for the `docker ps` command.

In addition to adding the `DockerInspectResponse` struct, we're also going to add a new method on our existing `Docker` struct. Let's call this method `Inspect` and it should take a string that represents the container ID we want it to inspect. Then, it should return a `DockerInspectResponse`. The body of the method is straightforward. It creates an instance of a Docker client called `dc`.

Then we call the client's `ContainerInspect` method, passing in a context `ctx` and a `containerID`. We check for an error and return it if we find one. Otherwise, we return a `DockerInspectResponse`.

Listing 9.2. The `Inspect` method calls the Docker API to get the authoritative state of a task's container.

```
func (d *Docker) Inspect(containerID string) DockerInspectRespons
        dc, _ := client.NewClientWithOpts(client.FromEnv)
        ctx := context.Background()
        resp, err := dc.ContainerInspect(ctx, containerID)
        if err != nil {
                log.Printf("Error inspecting container: %s\n", er
                return DockerInspectResponse{Error: err}
        }

        return DockerInspectResponse{Container: &resp}
}
```

Now that we've implemented the means to inspect a task, let's move on and use it in our worker.

9.4.2 Implementing task updates on the worker

In order for the worker to update the state of its tasks, we'll need to refactor it to use the new `Inspect` method we created on the `Docker` struct. To start, let's open the worker/worker.go file and add the `InspectTask` method seen below. This method takes a single argument `t` of type `task.Task`. It creates a task config `config`, then sets up an instance of the `Docker` type that will allow us to interact with the Docker daemon running on the worker. Finally, it calls the `Inspect` method, passing in the `ContainerID`.

Listing 9.3. The `InspectTask` method calls the new `Inspect` method on the `Docker` struct.

```
func (w *Worker) InspectTask(t task.Task) task.DockerInspectRespo
        config := task.NewConfig(&t)
        d := task.NewDocker(config)
        return d.Inspect(t.ContainerID)
}
```

Next, the worker will need to call its new `InspectTask` method. To do this,

let’s use the same pattern we’ve used in the past. We’ll create a public method called `UpdateTasks`, which will allow us to call to run in a separate goroutine. This method is nothing more than a wrapper that runs a continous loop and calls the private `updateTasks` method, which does all the heavy lifting.

Listing 9.4. The worker’s new `UpdateTasks` method serves as a wrapper around the new `updateTasks` method.

```
func (w *Worker) UpdateTasks() {
        for {
                log.Println("Checking status of tasks")
                w.updateTasks()
                log.Println("Task updates completed")
                log.Println("Sleeping for 15 seconds")
                time.Sleep(15 * time.Second)
        }
}
```

The `updateTasks` method performs a very simple algorithm. For each task in the worker’s datastore, it does the following:

1. call the `InspectTask` method to get the task’s state from the Docker daemon
2. verify the task is in `running` state
3. if it’s not in a `running` state, or not running at all, it sets the tasks' state to `failed`

The `updateTasks` method also performs one other operation. It sets the `HostPorts` field on the task. This allows us to see what ports the Docker daemon has allocated to the task’s running container.

Listing 9.5. The worker’s new updateTasks method handles calling the new `InspectTask` method, which results in updating the task’s state based on the state of its Docker container.

```
func (w *Worker) updateTasks() {
        for id, t := range w.Db {
                if t.State == task.Running {
                        resp := w.InspectTask(*t)
                        if resp.Error != nil {
                                fmt.Printf("ERROR: %v", resp.Erro
                        }
```

```
                        if resp.Container == nil {
                                log.Printf("No container for runn
                                w.Db[id].State = task.Failed
                        }

                        if resp.Container.State.Status == "exited
                                log.Printf("Container for task %s
                                w.Db[id].State = task.Failed
                        }

                        w.Db[id].HostPorts = resp.Container.Netwo
                }
        }
}
```

9.4.3 Healthchecks and restarts

We've said we will have the manager perform health checks for the tasks running in our orchestration system. But, how do we identify these health checks so the manager can call them? One simple way to accomplish this is to add a field called `HealthCheck` to the `Task` struct, and by convention we can use this new field to include a URL that the manager can call to perform a health check.

In addition to the `HealthCheck` field, let's also add a field called `RestartCount` to the `Tasks` struct. This field will be incremented each time the task is restarted, as we will see later in this chapter.

Listing 9.6. We add the `HealthCheck` and `RestartCount` fields to the existing `Task`struct.

```
type Task struct {
    // existing fields omitted
        HealthCheck   string
        RestartCount  int
}
```

The benefit of this approach to health checks is that it makes it the responsibility of the task to define what it means to be "healthy". Indeed, the definition of "healthy" can vary wildly from task to task. Thus, by having the task define its health check as a URL that can be called by the manager, all the manager has to do then is to call that URL. The result of calling the task's

health check URL then determines a task's health: if the call returns a 200 status, the task is healthy; otherwise it is not.

Now that we've implemented the necessary bits to enable our health check strategy, let's write the code necessary for the manager to make use of that work.

Let's start with the lowest level code first. Open the `manager/manager.go` file in your editor, and add the `checkTaskHealth` method seen below. This method implements the necessary steps that allow the manager to check the health of an individual task. It takes a single argument `t` of type `task.Task`, and it returns an error if the health check is not successful.

There are a couple things to note about this method. First, recall that when the manager schedules a task onto a worker, it adds an entry in its `TaskWorkerMap` field that maps the task's ID to the worker where it has been scheduled. That entry is a string and will be the IP address and port of the worker, e.g. `192.168.1.100:5555`. Thus, it's the address of the worker's API. The task will, of course, be listening on a different port from the worker API. Thus, it's necessary to get the task's port that the Docker daemon assigned to it when the task started, and we accomplish this by calling the `getHostPort` helper method. Then, using the worker's IP address, the port on which the task is listening, and the health check defined in the task's definition, the manager can build a URL like `http://192.168.1.100:49847/health`.

Listing 9.7. The manager's new `checkTaskHealth` method is responsible for calling a task's healthcheck URL.

```
func (m *Manager) checkTaskHealth(t task.Task) error {
        log.Printf("Calling health check for task %s: %s\n", t.ID

        w := m.TaskWorkerMap[t.ID] #1
        hostPort := getHostPort(t.HostPorts) #2
        worker := strings.Split(w, ":") #3
        url := fmt.Sprintf("http://%s:%s%s", worker[0], *hostPort
        log.Printf("Calling health check for task %s: %s\n", t.ID
        resp, err := http.Get(url) #5
        if err != nil {
                msg := fmt.Sprintf("Error connecting to health ch
                log.Println(msg)
                return errors.New(msg) #6
```

```
        }

        if resp.StatusCode != http.StatusOK { #7
                msg := fmt.Sprintf("Error health check for task %
                log.Println(msg)
                return errors.New(msg)
        }

        log.Printf("Task %s health check response: %v\n", t.ID, r

        return nil
}
```

Listing 9.8. The `getHostPort` method is a helper that returns the host port where the task is listening.

```
func getHostPort(ports nat.PortMap) *string {
        for k, _ := range ports {
                return &ports[k][0].HostPort
        }
        return nil
}
```

Now that our manager knows how to call individual task health checks, let's create a new method that will use that knowledge to operate on all the tasks in our system. It's important to note that we want the manager to check the health of tasks only in `Running` or `Failed` states. Tasks in `Pending` or `Scheduled` start are in the process of being started, so we don't want to attempt calling their health checks at this point. And, the `Completed` state is terminal, meaning the task has stopped normally and is in the expected state.

The process we'll use to check the health of individual tasks will involve iterating over the tasks in the manager's `TaskDb`. If a task is in the `Running` state, then it will call the task's health check endpoint and attempt to restart the task if the health check fails. If a task is in the `Failed` state, then there is no reason to call its health check, so we move on and attempt to restart the task. We can summarize this process like this:

- If the task is in `Running` state:
- Call the manager's `checkTaskHealth` method, which in turn will call the task's health check endpoint
- If the task's health check fails, attempt to restart the task

- If the task is in `Failed` state:
- attempt to restart the task

The above process is coded as you see below in the `doHealthChecks` method in listing X. Notice that we are only attempting to restart failed tasks if their `RestartCount` field is less than three. We are arbitrarily choosing to only attempt to restart failed tasks three times. If we were writing a production-quality system, we would likely do something smarter and much more sophisticated.

Listing 9.9. The manager's `doHealthChecks` method has primary responsibility for health checks.

```
func (m *Manager) doHealthChecks() {
        for _, t := range m.TaskDb {
        if t.State == task.Running && t.RestartCount < 3 {
                        err := m.checkTaskHealth(*t)
                        if err != nil {
                                if t.RestartCount < 3 {
                                        m.restartTask(t)
                                }
                        }
                } else if t.State == task.Failed && t.RestartCoun
                        m.restartTask(t)
                }
        }
}
```

The `doHealthChecks` method above calls the `restartTask` method seen here. Despite the number of lines involved, this code is fairly straightforward. Because our manager is naively attempting to restart the task on the same worker where the task was originally scheduled, it looks up that worker in its `TaskWorkerMap` using the task's `task.ID` field. Next, it's changes the task's state to `Scheduled`, and it increments the task's `RestartCount`. Then, it overwrites the existing task in the `TaskDb` datastore to ensure the manager has the correct state of the task. At this point, the rest of the code should look familiar. It creates a `task.TaskEvent` and adds the `task` to it, then marshals the `TaskEvent` into JSON. Using the JSON-encoded `TaskEvent`, it sends a POST request to the worker's API to restart the task.

Listing 9.10. The manager's new `restartTask` method is responsible for restarting tasks that have failed.

```
func (m *Manager) restartTask(t *task.Task) {
        // Get the worker where the task was running
        w := m.TaskWorkerMap[t.ID]
        t.State = task.Scheduled
        t.RestartCount++
        // We need to overwrite the existing task to ensure it ha
        // the current state
        m.TaskDb[t.ID] = t

        te := task.TaskEvent{
                ID:        uuid.New(),
                State:     task.Running,
                Timestamp: time.Now(),
                Task:      *t,
        }
        data, err := json.Marshal(te)
        if err != nil {
                log.Printf("Unable to marshal task object: %v.",
        }

        url := fmt.Sprintf("http://%s/tasks", w)
        resp, err := http.Post(url, "application/json", bytes.New
        if err != nil {
                log.Printf("Error connecting to %v: %v", w, err)
                m.Pending.Enqueue(t)
                return
        }

        d := json.NewDecoder(resp.Body)
        if resp.StatusCode != http.StatusCreated {
                e := worker.ErrResponse{}
                err := d.Decode(&e)
                if err != nil {
                        fmt.Printf("Error decoding response: %s\n
                        return
                }
                log.Printf("Response error (%d): %s", e.HTTPStatu
                return
        }

        newTask := task.Task{}
        err = d.Decode(&newTask)
        if err != nil {
                fmt.Printf("Error decoding response: %s\n", err.E
                return
        }
        log.Printf("%#v\n", t)
}
```

With the low-level details implemented, we can wrap our the necessary coding for the manager by writing the `DoHealthChecks` method seen below. This method will be used to run the manager's health checking functionality in a separate goroutine.

Listing 9.11. Like other wrapper methods, the `DoHealthChecks` method wraps the `doHealthChecks` method.

```
func (m *Manager) DoHealthChecks() {
        for {
                log.Println("Performing task health check")
                m.doHealthChecks()
                log.Println("Task health checks completed")
                log.Println("Sleeping for 60 seconds")
                time.Sleep(60 * time.Second)
        }
}
```

9.5 Putting it all together

In order to test our code and see it work, we'll need a task that implements a health check. Also, we'll want a way to trigger it to fail so our manager will attempt to restart it. We can use the echo service mentioned at the beginning of the chapter for this purpose. I've made it available as a Docker image that we can pull and run locally, thus saving us the time of writing additional code.

To run it, use this command:

```
$ docker run -p 7777:7777 --name echo timboring/echo-server:lates
```

The echo service implements three endpoints. Calling the root endpoint `/` with a POST and a JSON-encoded request body will simply echo a JSON request body back in a response body.

```
$ curl -X POST http://localhost:7777/ -d '{"Msg": "hello world"}'
{"Msg":"hello world"}
```

Calling the `/health` endpoint with a GET will return an empty body with a

`200 OK` response.

```
$ curl -v http://localhost:7777/health
*   Trying 127.0.0.1:7777...
* Connected to localhost (127.0.0.1) port 7777 (#0)
> GET /health HTTP/1.1
> Host: localhost:7777
> User-Agent: curl/7.83.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Sun, 12 Jun 2022 16:17:02 GMT
< Content-Length: 2
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
OK
```

Finally, calling the `/healthfail` endpoint with a GET will return an empty body with a `500 Internal Server Error` response.

```
$ curl -v http://localhost:7777/healthfail
*   Trying 127.0.0.1:7777...
* Connected to localhost (127.0.0.1) port 7777 (#0)
> GET /healthfail HTTP/1.1
> Host: localhost:7777
> User-Agent: curl/7.83.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 500 Internal Server Error
< Date: Sun, 12 Jun 2022 16:17:45 GMT
< Content-Length: 21
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
Internal server error
```

At this point, we can start up our worker and manager locally. We only need to make two tweaks to the code in the `main.go` file from chapter 8. The first is to call the new `UpdateTasks` method on our worker. The second is to call the new `DoHealthChecks` method on our manager. The rest of the code remains the same and results in the worker and manager starting up.

Listing 9.12. We only need to make two changes to the `main.go` file from chapter 8 in order to make use of the code written in this chapter.

```
func main() {
        w := worker.Worker{
                Queue: *queue.New(),
                Db:    make(map[uuid.UUID]*task.Task),
        }
        wapi := worker.Api{Address: whost, Port: wport, Worker: &

        go w.RunTasks()
        go w.CollectStats()
        *go w.UpdateTasks()*
        go wapi.Start()

        m := manager.New(workers)
        mapi := manager.Api{Address: mhost, Port: mport, Manager:

        go m.ProcessTasks()
        go m.UpdateTasks()
        *go m.DoHealthChecks()*

        mapi.Start()

}
```

When we start up the worker and manager, we should see familiar output like this.

```
2022/06/12 12:25:52 Sleeping for 15 seconds
2022/06/12 12:25:52 Collecting stats
2022/06/12 12:25:52 Checking for task updates from workers
2022/06/12 12:25:52 Checking worker localhost:5555 for task updat
2022/06/12 12:25:52 No tasks to process currently.
2022/06/12 12:25:52 Sleeping for 10 seconds.
2022/06/12 12:25:52 Processing any tasks in the queue
2022/06/12 12:25:52 No work in the queue
2022/06/12 12:25:52 Sleeping for 10 seconds
2022/06/12 12:25:52 Performing task health check
2022/06/12 12:25:52 Task health checks completed
2022/06/12 12:25:52 Sleeping for 60 seconds
2022/06/12 12:25:52 Task updates completed
2022/06/12 12:25:52 Sleeping for 15 seconds
```

We can verify the worker and manager are indeed working as expected by sending requests to their APIs.

```
# querying the worker API
$ curl localhost:5555/tasks
[]

# querying the manager API
$ curl localhost:5556/tasks
[]
```

As we'd expect, both the worker and manager return empty lists for their respective `/tasks` endpoints.

So let's create a task so the manager can start it up. To simplify the process, create a file called `task1.json` and add the following JSON to it:

Listing 9.13. We can store a task in JSON format in a file and pass the file to the curl command, thus saving us time and confusion.

```
{
    "ID": "a7aa1d44-08f6-443e-9378-f5884311019e",
    "State": 2,
    "Task": {
        "State": 1,
        "ID": "bb1d59ef-9fc1-4e4b-a44d-db571eeed203",
        "Name": "test-chapter-9.1",
        "Image": "timboring/echo-server:latest",
        "ExposedPorts": {
            "7777/tcp": {}
        },
        "PortBindings": {
            "7777/tcp": "7777"
        },
        "HealthCheck": "/health"
    }
}
```

Next, let's make a POST request to the manager using this JSON as a request body.

```
$ curl -v -X POST localhost:5556/tasks -d @task1.json
```

When we submit the task to the manager, we should see the manager and worker going through their normal paces to create the task. Ultimately, we should see the task in a running state if we query the manager.

```
$ curl http://localhost:5556/tasks|jq
[
  {
    "ID": "bb1d59ef-9fc1-4e4b-a44d-db571eeed203",
    "ContainerID": "fbdcb43461134fc20cafdfcdadc4cc905571c386908b1
    "Name": "test-chapter-9.1",
    "State": 2,
    "Image": "timboring/echo-server:latest",
    "Cpu": 0,
    "Memory": 0,
    "Disk": 0,
    "ExposedPorts": {
      "7777/tcp": {}
    },
    "HostPorts": {
      "7777/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "49155"
        },
        {
          "HostIp": "::",
          "HostPort": "49154"
        }
      ]
    },
    "PortBindings": {
      "7777/tcp": "7777"
    },
    "RestartPolicy": "",
    "StartTime": "0001-01-01T00:00:00Z",
    "FinishTime": "0001-01-01T00:00:00Z",
    "HealthCheck": "/health",
    "RestartCount": 0
  }
]
```

And, we should eventually see output from the manager showing it calling the task's health check.

```
2022/06/12 13:17:13 Performing task health check
2022/06/12 13:17:13 Calling health check for task bb1d59ef-9fc1-4
2022/06/12 13:17:13 Calling health check for task bb1d59ef-9fc1-4
2022/06/12 13:17:13 Task bb1d59ef-9fc1-4e4b-a44d-db571eeed203 hea
2022/06/12 13:17:13 Task health checks completed
```

This is good news. We can see that our health checking strategy is working. Well, at least in the case of a healthy task! What happens in the case of an unhealthy one?

To see what happens if a task's health check fails, let's submit another task to the manager. This time, we're going to set the task's health check endpoint to `/healthfail`.

Listing 9.14. The JSON definition for our second task, which includes a healthcheck that will result in a non-200 response.

```
{
    "ID": "6be4cb6b-61d1-40cb-bc7b-9cacefefa60c",
    "State": 2,
    "Task": {
        "State": 1,
        "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
        "Name": "test-chapter-9.2",
        "Image": "timboring/echo-server:latest",
        "ExposedPorts": {
            "7777/tcp": {}
        },
        "PortBindings": {
            "7777/tcp": "7777"
        },
        "HealthCheck": "/healthfail"
    }
}
```

If we watch the output in the terminal where out worker and manager are running, we should eventually see the call to this task's `/healthfail` endpoint return a non-200 response.

```
2022/06/12 13:37:30 Calling health check for task 21b23589-5d2d-4
2022/06/12 13:37:30 Calling health check for task 21b23589-5d2d-4
2022/06/12 13:37:30 Error health check for task 21b23589-5d2d-473
```

And, as a result of this health check failure, we should see the manager attempt to restart the task.

```
2022/06/12 13:37:30 Added task 21b23589-5d2d-4731-b5c9-a97e9832d0
```

This process should continue until the task has been restarted three times,

after which the manager will stop trying to restart the task, thus leaving it in `Running` state. We can see this by querying the manager's API and looking at the task's `State` and `RetryCount` fields.

```
{
    "ID": "21b23589-5d2d-4731-b5c9-a97e9832d021",
    "ContainerID": "acb37c0c2577461cae93c50b894eccfdbc363a6c51ea2
    "Name": "test-chapter-9.2",
    "State": 2,
    "Image": "timboring/echo-server:latest",
    // other fields omitted
    "RestartCount": 3
  }
```

In addition to restarting tasks when their health check fails, this strategy also works in the case where a task dies. For example, we can simulate the situation of a task dying by stopping the task's container manually using the `docker stop` command. Let's do this for the first task we created above, which should still be running.

```
2022/06/12 14:14:08 Performing task health check
2022/06/12 14:14:08 Calling health check for task bb1d59ef-9fc1-4
2022/06/12 14:14:08 Calling health check for task bb1d59ef-9fc1-4
2022/06/12 14:14:08 Error connecting to health check http://local
2022/06/12 14:14:08 Added task bb1d59ef-9fc1-4e4b-a44d-db571eeed2
```

And we should then see the container running again:

```
$ docker ps
CONTAINER ID   IMAGE                            CREATED          ST
1d75e69fa804   timboring/echo-server:latest   36 seconds ago   Up
```

And if we query the manager's API, we should see the task has a `RestartCount` of `1`:

```
$ curl http://localhost:5556/tasks|jq
[
  {
    "ID": "bb1d59ef-9fc1-4e4b-a44d-db571eeed203",
    "ContainerID": "1d75e69fa80431b39980ba605bccdb2e049d39d44afa3
    "Name": "test-chapter-9.1",
    "State": 2,
    "Image": "timboring/echo-server:latest",
    // other fields omitted
```

```
    "RestartCount": 1
  }
]
```

9.6 Summary

- Failures happen all the time, and the causes can be numerous.
- Handling failures in an orchestration system is complex. We implemented task-level health checking as a strategy to handle a small number of failure scenarios.
- An orchestration system can automate the process of recovering from task failures up to a certain point. Past a certain point, however, the best it can do is provide useful debugging information that helps administrators and application owners troubleshoot the problem further. (Note that we did not do this for our orchestration system. If you're curious, you can try to implement something like task logging.)

10 Implementing a more sophisticated scheduler

This chapter covers

- Describing the scheduling problem
- Defining the phases of scheduling
- Re-implementing the round robin scheduler
- Discussing the Enhanced PVM (E-PVM) concept and algorithm
- Implementing the E-PVM scheduler

We implemented a simple round-robin scheduler in chapter 7. Now, let's return and dig a little deeper into the general problem of scheduling and see how we might implement a more sophisticated scheduler.

10.1 The scheduling problem

Whether we realize it or not, the scheduling problem lives with us in our daily lives. In our homes, we have work to do like sweeping the floors, cooking meals, washing clothes, mowing the grass, and so on. Depending on the size of our family, we have 1 or more people to perform the necessary work. If you live by yourself, then you have a single worker, yourself. If you live with a partner, you have two workers. If you live with a partner and children, then you have 3 or more workers.

Now, how do we assign our house work to our workers? Do you take it all on yourself because your partner is taking the kids to soccer practice? Do you mow the grass and wash the clothes, while your partner will sweep the floors and cook dinner after returning from taking the kids to soccer practice? Do you take the kids to soccer practice while your partner cooks dinner, and when you return your oldest child will mow the grass and the youngest will do the laundry while your partner sweeps the floors?

In chapter 6, we described the scheduling problem using the scenario of a host seating customers at a busy restaurant on a Friday night. The host has six servers waiting on customers sitting at tables spread across the room. Each customer at each of those tables has different requirements. One customer might be there to have drinks and appetizers with a group of friends she hasn't seen in a while. Another customer might be there for a full dinner, complete with an appetizer and desert. Yet another customer might have strict dietary requirements and only eats plant-based food.

Now, a new customer walks in. It's a family of four: two adults, two teenage children. Where does the host seat them? Do they place them at the table in the section being served by John, who already has three tables with four customers each? Do they place them at the table in Jill's section, who has six tables with a single customer each? Or, do they place them in Willie's section, who has a single table with three customers?

The same scheduling problem also exists in our work lives. Most of us work on a team, and we have work that needs to get done: writing documentation for a new feature or a new piece of infrastructure; fixing a critical bug that a customer reported over the weekend; drafting team goals for the next quarter. How do we divvy up this work amongst ourselves?

As we can from the above examples, scheduling is all around us.

10.2 Scheduling considerations

When we implemented the round-robin scheduler in chapter 7, we didn't take much into consideration. We just wanted a simple implementation of a scheduler that we could implement quickly so we could focus on other areas of our orchestration system.

There are, however, a wide range of things to consider if we take the time. What goals are we trying to achieve?

- Seat customers as quickly as possible to avoid a large queue of customers having to wait
- Distribute customers evenly across our servers so they get the best

service
- Get customers in and out as quickly as possible because we want high volume.

The same considerations exist in an orchestration system. Instead of seating customers at tables, we're placing tasks on machines.

- Do we want the task to be placed and running as quickly as possible?
- Do we want to place the task on a machine that is best capable of meeting the unique needs of the task?
- Do we want to place the task on a machine that will distribute the load evenly across all of our workers?

10.3 Scheduler interface

Unfortunately, there is no one-size-fits-all approach to scheduling. How we schedule tasks depends on the goals we want to achieve. For this reason, most orchestrators support multiple schedulers. Kubernetes achieves this through scheduling *Profiles* (see https://kubernetes.io/docs/reference/scheduling/config/), while Nomad achieves it through four scheduler types (see https://developer.hashicorp.com/nomad/docs/schedulers).

Like Kubernetes and Nomad, we want to support more than one type of scheduler. We can accomplish this by using an interface. In fact, we already defined such an interface in chapter 2.

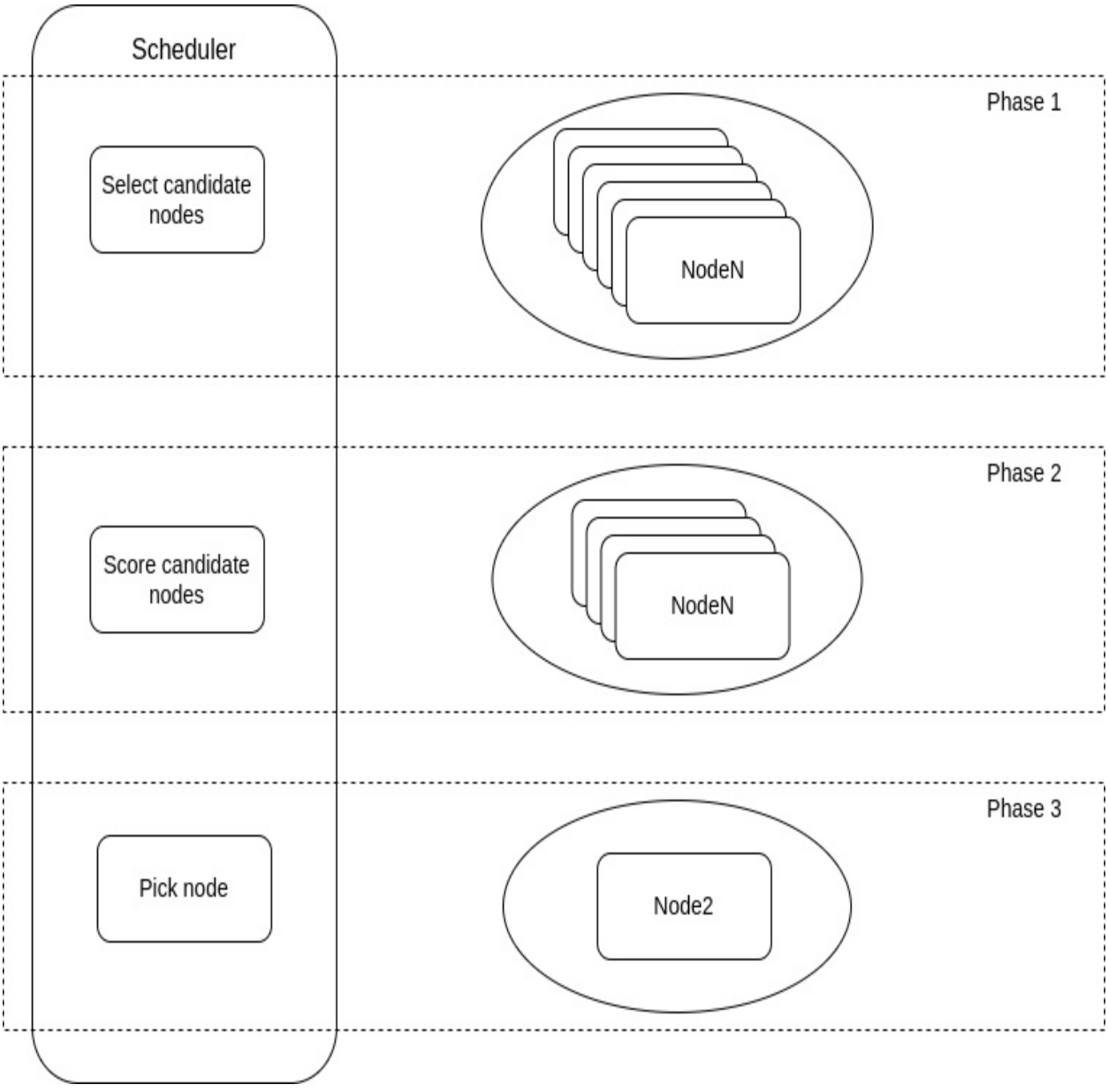
```
type Scheduler interface {
    SelectCandidateNodes()
    Score()
    Pick()
}
```

Our interface is simple. It has three methods:

- SelectCandidateNodes
- Score
- Pick

We can think of these methods as the different phases of the scheduling problem, as seen in figure 10.1.

Figure 10.1. The scheduling problem can be broken down into three phases: selecting candidate nodes, scoring candidate nodes, and, finally, picking one of the nodes



Using just these three methods, we can implement any number of schedulers. Before we dive into writing any new code, however, let's revise our `Scheduler` interface with a few more details.

To start, we want our `SelectCandidateNodes` method to accept a task and a list of nodes. As we will soon see, this method acts as a filter early in the scheduling process, reducing the number of possible workers to only those that we are confident can run the task. For example, if our task needs 1 gig of disk space because we haven't taken the time to reduce the size of our Docker image, then we only want to consider scheduling the task onto workers that have at least 1 gig of disk space available to download our image. As a result, `SelectCandidateNodes` returns a list of nodes that will, at a minimum, meet the resource requirements of the task.

Next, we want our `Score` method to also accept a task and list of nodes as parameters. This method performs the heavy lifting. Depending on the scheduling algorithm we're implementing, this method will assign a score to each of the candidate nodes it receives. Then, it returns a map[string]float64, where the map key is the name of the node and the value is its score.

Finally, our `Pick` method needs to accept a map of scores, i.e. the output of the `Score` method, and a list of candidate nodes. Then, it picks the node with the "best" score. The definition of "best" is left as an implementation detail.

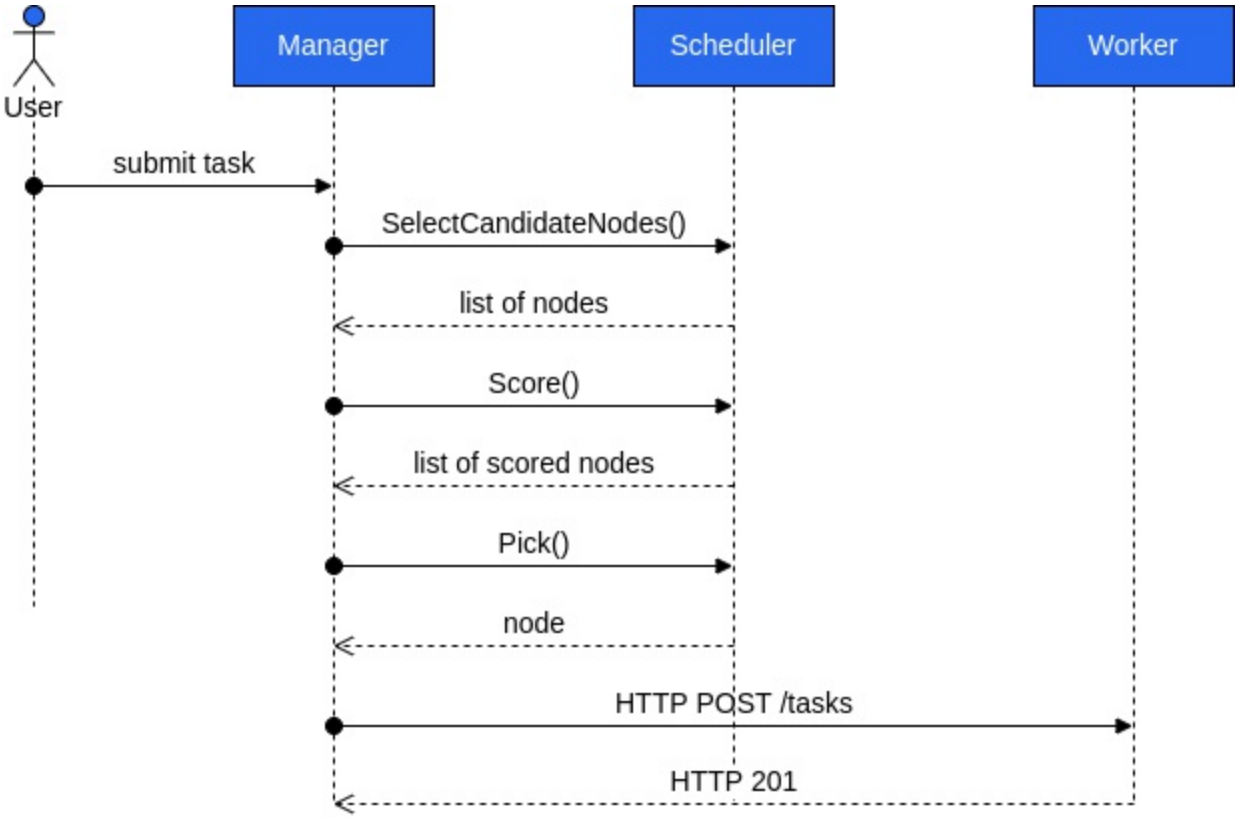
Listing 10.1. The updated `Scheduler` interface includes method parameters and return values.

```
type Scheduler interface {
        SelectCandidateNodes(t task.Task, nodes []*node.Node) []*
        Score(t task.Task, nodes []*node.Node) map[string]float64
        Pick(scores map[string]float64, candidates []*node.Node)
}
```

10.4 Adapting the round-robin scheduler to the scheduler interface

Since we have already implemented a round-robin scheduler, let's adapt that code to our scheduler interface. The sequence diagram in figure 10.2 illustrates how our manager will interact with the `Scheduler` interface to select a node for running a task.

Figure 10.2. Sequence diagram showing the interactions between the manager, scheduler, and worker.



Let's start by opening the `scheduler.go` file and creating the `RoundRobin` struct seen in listing 10.2. Our struct has two fields: `Name`, which allows us to give it a descriptive name, and `LastWorker`, which will take over the role of the field with the same name from the `Manager` struct.

Listing 10.2. The round-robin scheduler can be adapted to the `Scheduler` interface by defining a `RoundRobin` struct.

```
type RoundRobin struct {
        Name       string
        LastWorker int
}
```

Next, let's implement the `SelectCandidateNodes` method for our round-robin scheduler. Because we are adapting our original implementation, not improving upon it, we're going to simply return the list of nodes that are passed in as one of the two method parameters. While it might seem a bit silly for this method to just return what it received, the `RoundRobin` scheduler does need to implement this method in order to meet the contract of the `Scheduler` interface.

Listing 10.3. The `SelectCandidateNodes` method for the round-robin scheduler simply returns the nodes it receives without filtering out any.

```
func (r *RoundRobin) SelectCandidateNodes(t task.Task, nodes []*n
        return nodes
}
```

Now, let's implement the `Score` method for our round-robin implementation. Here, we're effectively taking the code from the existing manager's `SelectWorker` method and pasting it into the `Score` method. We do, however, need to make a few modifications.

First, we define the `nodeScores` variable, which is of type `map[string]float64`. This variable will hold the scores we assign to each node. Depending on the number of nodes we're using, the resulting map will look something like this:

```
{
    "node1": 1.0,
    "node2": 0.1,
    "node3": 1.0,
}
```

Second, we iterate over the list of `nodes` that are passed in to the method and build our map of `nodeScores`. Notice that our method of scoring is not that sophisticated. We check if the index is equal to the `newWorker` variable, and if it is then we assign the node a score of `0.1`. If the index is not equal to the `newWorker`, then we give it a score of `1.0`. Once we have built our map of `nodeScores`, we return it.

Listing 10.4. The `Score` method adapts the original code from the manager's `SelectWorker` method to the `Score` method of the `Scheduler` interface.

```
func (r *RoundRobin) Score(t task.Task, nodes []*node.Node) map[s
        nodeScores := make(map[string]float64) #1

        var newWorker int #2
        if r.LastWorker+1 < len(nodes) {
                newWorker = r.LastWorker + 1
                r.LastWorker++
        } else {
                newWorker = 0
```

```
                r.LastWorker = 0
        }

        for idx, node := range nodes { #3
                if idx == newWorker {
                        nodeScores[node.Name] = 0.1
                } else {
                        nodeScores[node.Name] = 1.0
                }
        }

        return nodeScores #4
}
```

With the `SelectCandidateNodes` and `Score` methods implemented, let's turn our attention to the final method of the `Scheduler` interface, the `Pick` method. As its name suggests, this method picks the "best" node to run a task. It accepts a `map[string]float64`, which will be the scores returned from the `Score` method above. It also accepts a list of candidate nodes. It returns a single type of a pointer to a `node.Node`.

For the purposes of the round-robin implementation, the "best" score is the lowest score. So, if we had a list of three nodes with scores of 0.1, 1.0, and 1.0, the node with the 0.1 score will be selected.

Listing 10.5. The round-robin scheduler's `Pick` method return the node with the lowest score.

```
func (r *RoundRobin) Pick(scores map[string]float64, candidates [
        var bestNode *node.Node
        var lowestScore float64
        for idx, node := range candidates { #1
                if idx == 0 { #2
                        bestNode = node
                        lowestScore = scores[node.Name]
                        continue
                }

                if scores[node.Name] < lowestScore { #3
                        bestNode = node
                        lowestScore = scores[node.Name]
                }
        }

        return bestNode #4
```

```
}
```

With the implementation of the `Pick` method, we have completed adapting the round-robin scheduler to the `Scheduler` interface. Now, let's see how we can use it.

10.5 Using the new scheduler interface

In order to use the new `Scheduler` interface, there are a few changes we need to make to the manager. There are three types of changes to do:

- Adding new fields to the `Manager` struct
- Modifying the `New` helper function in the `manager` package
- Modifying several of the manager's methods to use the scheduler

10.5.1 Adding new fields to the `Manager` struct

The first set of changes to make are to add two new fields to our `Manager` struct. As you can see below in listing 10.6, the first is a field named `WorkerNodes`. This field is a slice of pointers of `node.Node`. It will hold instances of each worker node. The second field we need to add is `Scheduler`, which is our new interface type `scheduler.Scheduler`. As you will see later, defining the `Scheduler` field type as the `Scheduler` interface allows the manager to use any scheduler that implements the interface.

Listing 10.6. The `WorkerNodes` and `Scheduler` fields are new to the `Manager` struct.

```
type Manager struct {
    // previous code not shown

        WorkerNodes   []*node.Node
        Scheduler     scheduler.Scheduler
}
```

10.5.2 Modifying the `New` helper function

The second set of changes involve modifying the `New` function in the `manager` package. The first of these changes is to add the `schedulerType` parameter to

the function signature. This parameter will allow us to create a manager with one of the concrete scheduler types, starting with the `RoundRobin` type. The next change to make involves adding an `error` to the types returned from the function. The changes to the function signature can be seen in listing 10.7.

Listing 10.7. The `New` helper function is modified to take a new argument, `schedulerType` and return an `error` in addition to a pointer to the `Manager`.

```
func New(workers []string, schedulerType string) (*Manager, error
```

The next change to the function happens in the body. We define the variable `nodes` to hold a slice of pointers to the `node.Node` type. We're going to perform this work inside the existing loop over the `workers` slice. Inside this loop, we create a node by calling the `node.NewNode` function, passing it the name of the worker, the address for the worker's API (e.g. http://192.168.33.17:5556), and the node's role. All three values passed in to the `NewNode` function are strings. Once we have created the node, we add it to the slice of `nodes` by calling the built-in `append` function.

After creating the list of `nodes`, we can move on to the next-to-last change in the function body. Depending on the `schedulerType` passed in, we need to create a scheduler of the appropriate type. To do this, we create the variable `s` to hold the scheduler. Then, we use a `switch` statement to initialize the appropriate scheduler. We start out with only a single case to support the "roundrobin" scheduler. If `schedulerType` is not "roundrobin", then we return an error with an informative message.

The final changes to the function are simple. We need to add our list of `nodes` and our scheduler `s` to the `Manager` that we're returning at the end of the function. We do this by adding the slice of `nodes` to the `WorkerNodes` field, and the scheduler `s` to the `Scheduler` field.

Listing 10.8. Changes to the `New` helper function to use the new `Scheduler` interface.

```
func New(workers []string, schedulerType string) (*Manager, error
    // previous code not shown

        var nodes []*node.Node
        for worker := range workers {
```

```
                workerTaskMap[workers[worker]] = []uuid.UUID{}

                nAPI := fmt.Sprintf("http://%v", workers[worker])
                n := node.NewNode(workers[worker], nAPI, "worker"
                nodes = append(nodes, n)
        }

        var s scheduler.Scheduler
        switch schedulerType {
        case "roundrobin":
                s = &scheduler.RoundRobin{Name: "roundrobin"}
    default:
        return nil, fmt.Errorf("unsupported scheduler type")
        }

        return &Manager{
                Pending:       *queue.New(),
                Workers:       workers,
                TaskDb:        taskDb,
                EventDb:       eventDb,
                WorkerTaskMap: workerTaskMap,
                TaskWorkerMap: taskWorkerMap,
                WorkerNodes:   nodes,
                Scheduler:     s,
        }
}
```

With the changes to the `New` function above, we can now create our manager with different types of schedulers. But, we still have more work to do on our manager before it can actually use the scheduler.

The next piece of the `Manager` type we need to change is the `SelectWorker` method. We're going to scrap the previous implementation of this method and replace it. Why? Because the previous implementation was specifically geared toward the round-robin scheduling algorithm. With the creation of the `Scheduler` interface and the `RoundRobin` implementation of that interface, we need to refactor the `SelectWorker` method to operate on the scheduler interface.

As you can in listing 10.9, the `SelectWorker` method becomes more straightforward. It does the following:

- Call the manager's `Scheduler.SelectCandidateNodes` method, passing

it the task `t` and the slice of nodes in the manager's `WorkerNodes` field. If the call to `SelectCandidateNodes` results in the `candidates` variable being nil, then we return an error.
- Call the manager's `Scheduler.Score` method, passing it the task `t` and slice of `candidates`.
- Call the manager's `Scheduler.Pick` method, passing it the `scores` from the previous step and the slice of `candidates`.
- Return the `selectedNode`.

Listing 10.9. The `SelectWorker` method now uses the `Scheduler` interface to select a worker.

```
func (m *Manager) SelectWorker(t task.Task) (*node.Node, error) {
        candidates := m.Scheduler.SelectCandidateNodes(t, m.Worke
        if candidates == nil {
                msg := fmt.Sprintf("No available candidates match
                err := errors.New(msg)
                return nil, err
        }
        scores := m.Scheduler.Score(t, candidates)
        selectedNode := m.Scheduler.Pick(scores, candidates)

        return selectedNode, nil
}
```

Now, let's move on to the `SendWork` method of the manager. In the original version, the beginning of the method looked like that in listing 10.10 below. Notice that we called the `SelectWorker` method first, and we didn't pass it a task. We need to rework the beginning of this method to account for the changes we made to the `SelectWorker` method above.

Listing 10.10. The original `SendWork` method called `SelectWorker` at the beginning of the method.

```
func (m *Manager) SendWork() {
        if m.Pending.Len() > 0 {
                w := m.SelectWorker()

                e := m.Pending.Dequeue()
                te := e.(task.TaskEvent)
                t := te.Task
                log.Printf("Pulled %v off pending queue", t)

                m.EventDb[te.ID] = &te
```

```
                m.WorkerTaskMap[w] = append(m.WorkerTaskMap[w], t
                m.TaskWorkerMap[t.ID] = w
```

Instead of calling the `SelectWorker` method first, we now want to pop a task off the manager's pending queue as the first step. Then we do some accounting work, notably adding the `task.TaskEvent te` to the manager's `EventDb` map. It's only after pulling a task off the queue and doing the necessary accounting work that we then call the new version of `SelectWorker`. Moreover, we pass the task `t` to the new `SelectWorker` method.

Listing 10.11. The new `SendWork` method re-orders the steps in the process of sending a task to a worker.

```
func (m *Manager) SendWork() {
        if m.Pending.Len() > 0 {
                e := m.Pending.Dequeue()
                te := e.(task.TaskEvent)
                m.EventDb[te.ID] = &te
                log.Printf("Pulled %v off pending queue", te)

                t := te.Task
                w, err := m.SelectWorker(t)
                if err != nil {
                        log.Printf("error selecting worker for ta
                }
```

One important thing to note about the changes above: the type returned from the new implementation of `SelectWorker` is no longer a string. `SelectWorker` now returns a type of `node.Node`. To make this more concrete, the old version of `SelectWorker` returned a string that looked like `192.168.13.13:1234`. So, we need to make a few minor adjustments throughout the remainder of the `SendWork` method to replace any usage of the old string value held in the variable `w` with the value of the node's `Name` field. Listing 10.12 below shows the affected lines that need to be changed.

Listing 10.12. The `w` variable has changed from a string to a `node.Node` type, so we need to use the `w.Name` field.

```
m.WorkerTaskMap[w.Name] = append(m.WorkerTaskMap[w.Name], te.Task
m.TaskWorkerMap[t.ID] = w.Name
```

```
url := fmt.Sprintf("http://%s/tasks", w.Name)
}
```

With that, we've completed all the necessary changes to our manager. Well, sort of.

10.6 Did you notice the bug?

Up until this chapter, we've been working with a single instance of the worker. This choice made it easier for us to focus on the bigger picture. In the next section, however, we're going to modify our `main.go` program to start three workers.

There is a problem lurking in the manager's `SendWork` method. Notice that it's popping a task off its queue, then selecting a worker where it will send that task. What happens, however, when the task popped off the pending queue is for an existing task? The most obvious case for this behavior is stopping a running task. In such a case, the manager already knows about the running task and the associated task event, so we shouldn't create new ones. Instead, we need to check for an existing task and update it as necessary.

Our code in the previous chapters was working by chance. Since we were running a single worker, the `SelectWorker` method only had one choice when it encountered a task intended to stop a running task. Since we're now running three workers, there is a 67 percent chance that the existing code will select a worker where the existing task to be stopped is NOT running.

Let's fix this problem!

To start, let's introduce a new method to our manager, called `stopTask`. This method takes two arguments: a `worker` of type string, and a `taskID` also of type string. From the name of the method and the names of the parameters it's obvious what the method will do. It uses the `worker` and `taskID` arguments to build a URL to the worker's `/tasks/{taskID}` endpoint. Then it creates a request by calling the `NewRequest` function in the `http` package. Next, it executes the request.

Listing 10.13. The new stopTask method.

```
func (m *Manager) stopTask(worker string, taskID string) {
        client := &http.Client{}
        url := fmt.Sprintf("http://%s/tasks/%s", worker, taskID)
        req, err := http.NewRequest("DELETE", url, nil)
        if err != nil {
                log.Printf("error creating request to delete task
                return
        }

        resp, err := client.Do(req)
        if err != nil {
                log.Printf("error connecting to worker at %s: %v"
                return
        }

        if resp.StatusCode != 204 {
                log.Printf("Error sending request: %v", err)
                return
        }

        log.Printf("task %s has been scheduled to be stopped", ta
}
```

Now, let's use the `stopTask` method by calling it from the `SendWork` method. We're going to add new code near the beginning of the first `if` statement, which checks the length of the manager's `Pending` queue. Just after the log statement that prints, "Pulled %v off pending queue", add a newline and enter the code after the `// new code` comment seen in listing 10.14.

Listing 10.14. Checking for existing tasks and calling the new stopTask method.

```
// existing code
if m.Pending.Len() > 0 {
        e := m.Pending.Dequeue()
        te := e.(task.TaskEvent)
        m.EventDb[te.ID] = &te
        log.Printf("Pulled %v off pending queue", te)

    // new code
        taskWorker, ok := m.TaskWorkerMap[te.Task.ID] #1
        if ok {
                persistedTask := m.TaskDb[te.Task.ID] #2
                if te.State == task.Completed && task.ValidStateT
                        m.stopTask(taskWorker, te.Task.ID.String(
                        return
```

```
            }

            log.Printf("invalid request: existing task %s is
            return
    }
```

10.7 Putting it all together

Now that we've made the changes to the manager so it can make use of the new scheduler interface, we're ready to make several changes to our `main.go` program. As I mentioned above, in previous chapters we used a single worker. That choice was mostly one of convenience. Given that we've implemented a more sophisticated `Scheduler` interface, however, it'll be more interesting to start up several workers. This will better illustrate how our scheduler works.

The first change to make in our main program is to create three workers. To do this, we take the same approach as in previous chapters, but repeat it three time to create workers `w1`, `w2`, and `w3` as seen in listing 10.15. After creating each worker, we then create an API for it. Notice that for the first worker, we use the existing `wport` variable for the `Port` field of the `API`. Then, in order to start multiple APIs we increment the value of the `wport` variable so each API has a unique port to run on. This saves us from having to specify three different variables when we run the program from the commandline.

Listing 10.15. Each worker is represented by a `worker.Worker` type and a `worker.Api` type.

```
    w1 := worker.Worker{
            Queue: *queue.New(),
            Db:    make(map[uuid.UUID]*task.Task),
    }
    wapi1 := worker.Api{Address: whost, Port: wport, Worker:

    w2 := worker.Worker{
            Queue: *queue.New(),
            Db:    make(map[uuid.UUID]*task.Task),
    }
    wapi2 := worker.Api{Address: whost, Port: wport + 1, Work

    w3 := worker.Worker{
            Queue: *queue.New(),
```

```
                Db:    make(map[uuid.UUID]*task.Task),
        }
        wapi3 := worker.Api{Address: whost, Port: wport + 2, Work
```

Now that we have our three workers and their APIs, let's start everything up. The process is the same as in past chapters, just doing it once for each worker/API combo.

Listing 10.16. We start up each worker in the same way as we did for a single worker.

```
go w1.RunTasks()
go w1.UpdateTasks()
go wapi1.Start()

go w2.RunTasks()
go w2.UpdateTasks()
go wapi2.Start()

go w3.RunTasks()
go w3.UpdateTasks()
go wapi3.Start()
```

The next change is to build a slice that contains all three of our workers.

```
workers := []string{
        fmt.Sprintf("%s:%d", whost, wport),
        fmt.Sprintf("%s:%d", whost, wport+1),
        fmt.Sprintf("%s:%d", whost, wport+2),
}
```

Now, we need to update the existing call to the manager package's `New` function to specify the type of scheduler we want the manager to use. As you can see below, we're going to start by using the "roundrobin" scheduler.

```
func main() {
        m := manager.New(workers, "roundrobin")
}
```

With these changes, we can now start up our `main` program. Notice that we start up in the same way, by passing in the necessary environment variables for the worker and manager and then using the command `go run main.go`. Also notice that the output we see looks the same as it has. We see that the program is starting up the worker; then it starts up the manager; then it starts

cycling through checking for new tasks, collecting stats, checking the status of tasks, and attempting to update any existing tasks.

```
$ CUBE_WORKER_HOST=localhost CUBE_WORKER_PORT=5556 CUBE_MANAGER_H
Starting Cube worker
Starting Cube manager
2022/11/12 11:28:48 No tasks to process currently.
2022/11/12 11:28:48 Sleeping for 10 seconds.
2022/11/12 11:28:48 Collecting stats
2022/11/12 11:28:48 Checking status of tasks
2022/11/12 11:28:48 Task updates completed
2022/11/12 11:28:48 Sleeping for 15 seconds
2022/11/12 11:28:48 Processing any tasks in the queue
2022/11/12 11:28:48 No work in the queue
2022/11/12 11:28:48 Sleeping for 10 seconds
2022/11/12 11:28:48 Checking for task updates from workers
2022/11/12 11:28:48 Checking worker localhost:5556 for task updat
2022/11/12 11:28:48 Checking worker localhost:5557 for task updat
2022/11/12 11:28:48 Checking worker localhost:5558 for task updat
2022/11/12 11:28:48 Performing task health check
2022/11/12 11:28:48 Task health checks completed
2022/11/12 11:28:48 Sleeping for 60 seconds
```

Next, let’s send a task to our manager. We’ll use the same command we have in past chapters to start up an instance of the echo server. The output from the `curl` command looks like what we’re used to seeing from previous chapters.

```
$ curl -X POST localhost:5555/tasks -d @task1.json
{
    "ID":"bb1d59ef-9fc1-4e4b-a44d-db571eeed203",
    "ContainerID":"",
    "Name":"test-chapter-9.1",
    "State":1,
    "Image":"timboring/echo-server:latest",
    "Cpu":0,
    "Memory":0,
    "Disk":0,
    "ExposedPorts": {
        "7777/tcp": {}
    },
    "HostPorts":null,
    "PortBindings": {
        "7777/tcp":"7777"
    },
    "RestartPolicy":"",
    "StartTime":"0001-01-01T00:00:00Z",
```

```
    "FinishTime":"0001-01-01T00:00:00Z",
    "HealthCheck":"/health",
    "RestartCount":0
}
```

After sending the above task to the manager, we should see something like the following in the output of our main program. This should look familiar. The manager is checking its pending queue for tasks. It finds the task we sent it using the above `curl` command.

```
2022/11/12 11:40:18 Processing any tasks in the queue
2022/11/12 11:40:18 Pulled {a7aa1d44-08f6-443e-9378-f5884311019e
```

The following output is from the worker. It shows that the manager selected it when it called its `SelectWorker` method, which calls the `SelectCandidateNodes`, `Score`, and `Pick` methods on the scheduler.

```
Found task in queue: {bb1d59ef-9fc1-4e4b-a44d-db571eeed203  test-
```

Once the task is running, we can see the manager check it for any updates.

```
2022/11/12 11:40:33 Checking for task updates from workers
2022/11/12 11:40:33 Checking worker localhost:5556 for task updat
2022/11/12 11:40:33 [manager] Attempting to update task bb1d59ef-
2022/11/12 11:40:33 Task updates completed
```

At this point, we can start the other two tasks we've been using in past chapters. Using the `RoundRobin` scheduler, you should notice the manager selecting each of the other two workers in succession.

So, we can see that the round-robin scheduler works. Now, let's implement a second scheduler.

10.8 The E-PVM scheduler

The next type of scheduler we're going to implement is more sophisticated than our round-robin scheduler. For our new scheduler, our goal is to spread the tasks across our cluster of worker machines so we minimize the CPU load of each node. In other words, we would rather each node in our cluster do some work, and have overhead for any bursts of work.

10.8.1 The theory

In order to accomplish our goal of spreading the load across our cluster, we're going to use an opportunity cost approach to scoring our tasks. It is one of the approaches that Google used for its Borg orchestrator in its early days, and is based on the work presented in the paper "An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster" (https://www.cnds.jhu.edu/pub/papers/mosix.pdf). According to the paper, "the key idea…is to convert the total usage of several heterogeneous resources…into a single homogeneous 'cost'. Jobs are then assigned to the machine where they have the lowest cost." The "heterogenous resources" are CPU and memory. The authors call this method "Enhanced PVM" (where *PVM* stands for "Parallel Virtual Machine").

The main idea here is that when a new task enters the system and needs to be scheduled, this algorithm will calculate a `marginal_cost` for each machine in our cluster. What does *marginal cost* mean? If each machine has a "homogeneous cost" that represents the total usage of all its resources, then the *marginal cost* is the amount that "homogeneous cost" will increase if we add a new task to its workload.

The paper provides us the pseudocode for this algorithm, seen in listing 10.15 below. If the `marginal_cost` of assigning the job to a machine is less than the `MAX_COST`, then assign we assign the machine to `machine_pick`. Once we've iterated through our list of machines, `machine_pick` will contain the machine with the lowest marginal cost. We will slightly modify our implementation of this pseudocode to fit our own purposes.

Listing 10.17. Pseudocode describing the algorithm used in the Enhanced PVM scheduling method.

```
max_jobs = 1;

while () {
  machine_pick = 1; cost = MAX_COST
    repeat {} until (new job j arrives)
    for (each machine m) {
      marginal_cost = power(n, percentage memory utilization on m
      power(n, (jobs on m + 1/max_jobs) - power(n, memory use on
```

```
      if (marginal_cost < cost) { machine_pick = m; }
   }

  assign job to machine_pick;
  if (jobs on machine_pick > max_jobs) max_jobs = max_jobs * 2;
}
```

10.9 In practice

Implementing our new scheduler, which we'll call the E-PVM scheduler, will follow a path similar to the one we used to adapt the round-robin algorithm to our Scheduler interface. We start by defining the `Epvm` struct below. Notice that we're only defining a single field, `Name`, because we don't need to keep track of the last selected node like we did for the round-robin scheduler.

```
type Epvm struct {
        Name string
}
```

Next, we implement the `SelectCandidateNodes` method of the E-PVM scheduler. Unlike the round-robin scheduler, in this version of `SelectCandidateNodes` we do actually attempt to narrow the list of potential candidates. We do this by checking that the resources the task is requesting are less than the resources the node has available. For our purposes, we're only checking disk, because we want to ensure the selected node has the available disk space to download the task's Docker image.

Listing 10.18. The `Epvm` scheduler's `SelectCandidateNodes` method filters out any nodes that can't the task's disk requirements.

```
func (e *Epvm) SelectCandidateNodes(t task.Task, nodes []*node.No
        var candidates []*node.Node
        for node := range nodes {
        if checkDisk(t, nodes[node].Disk-nodes[node].DiskAllocate
                        candidates = append(candidates, nodes[nod
        }
        }

        return candidates
}
```

```
func checkDisk(t task.Task, diskAvailable int64) bool {
        return t.Disk <= diskAvailable
}
```

Now, let's dive in to the meat of the E-PVM scheduler. It's time to implement the `Score` method based on the E-PVM pseudocode from above.

We start by defining a couple of variables that we'll use later in the method. The first variable we define is `nodeScores`, which is a type of `map[string]float64` and will hold the scores of each node. Next, we define the `maxJobs` variable. We are randomly setting it to the value of `4.0`, meaning each node can handle 4 tasks at most. I chose this value because I initially developed the code for this book using a cluster of several Raspberry Pis, and it seemed like a reasonable guess of how many tasks each Pi could handle. In a production system, we would tune this value based on an analysis of observed metrics from our running production system.

The next step is to iterate over each of our `nodes` passed in to the method and calculate the marginal cost of assigning the task to the node. This process involves eight steps:

- Calculate the node's current CPU usage
- Calculate the node's current CPU load
- Calculate the node's allocated memory
- Calculate the node's percentage of memory allocated
- Calculate the percentage of memory that would be allocated if the task were assigned to it
- Calculate the memory cost of adding the task to the node
- Calculate the CPU cost of adding the task to the node
- Add the memory and CPU costs to get the marginal cost of adding the task to the node

To calculate the node's current CPU usage, we use the `calculateCpuUsage` helper function defined later. Then, we call the `calculateLoad` helper function. This function takes two parameters, `usage` and `capacity`. The `usage` value we get from the previous call to `calculateCpuUsage`, and for the capacity we use a fraction of what we think our max load will be. This definition of usage comes from the E-PVM paper, which assumes that the

maximum possible load is "the smallest integer power of two greater than the largest load we have seen at any given time." Again, given that I originally developed this code using Raspberry Pis, and only three of them at that, I guessed that the highest load seen on any of the nodes was 80%.

Listing 10.19. The E-PVM scheduler's `Score` has the same signature as the one in the RoundRobin scheduler, but it calculates scores in a more complicated way.

```
func (e *Epvm) Score(t task.Task, nodes []*node.Node) map[string]
        nodeScores := make(map[string]float64)
        maxJobs := 4.0

        for _, node := range nodes {
                cpuUsage := calculateCpuUsage(node)
                cpuLoad := calculateLoad(cpuUsage, math.Pow(2, 0.

                memoryAllocated := float64(node.Stats.MemUsedKb()
                memoryPercentAllocated := memoryAllocated / float

                newMemPercent := (calculateLoad(memoryAllocated +

                memCost := math.Pow(LIEB, newMemPercent) + math.P

                cpuCost := math.Pow(LIEB, cpuLoad) + math.Pow(LIE

                nodeScores[node.Name] = memCost + cpuCost
                        nodeScores[node.Name] = marginalCost
                }
        }
        return nodeScores
}
```

Our score method uses two helper functions to calculate CPU usage and load. The first of these helpers, `calculateCpuUsage`, is itself a multi-step process. The code for this function is based on the algorithm presented in this Stack Overflow post, https://stackoverflow.com/a/23376195. I won't go into more details about this algorithm, because the post does a good job of covering the topic. So I'd urge you to read it if you are interested.

Listing 10.20. The `calculateCpuUsage` helper function calculates the CPU usage as a `float64`.

```
func calculateCpuUsage(node *node.Node) *float64 {
        stat1 := getNodeStats(node)
```

```
        time.Sleep(3 * time.Second)
        stat2 := getNodeStats(node)

        stat1Idle := stat1.CpuStats.Idle + stat1.CpuStats.IOWait
        stat2Idle := stat2.CpuStats.Idle + stat2.CpuStats.IOWait

        stat1NonIdle := stat1.CpuStats.User + stat1.CpuStats.Nice

        stat2NonIdle := stat2.CpuStats.User + stat2.CpuStats.Nice

        stat1Total := stat1Idle + stat1NonIdle
        stat2Total := stat2Idle + stat2NonIdle

        total := stat2Total - stat1Total
        idle := stat2Idle - stat1Idle

        var cpuPercentUsage float64
        if total == 0 && idle == 0 {
                cpuPercentUsage = 0.00
        } else {
                cpuPercentUsage = (float64(total) - float64(idle)
        }
        return &cpuPercentUsage
}
```

Note that this function is using a second helper function, `getNodeStats`. This function, seen in listing 10.21, is calling the `/stats` endpoint on the worker node and retrieving the worker's stats at that point in time.

Listing 10.21. The `getNodeStats` helper function returns the stats for a given node.

```
func getNodeStats(node *node.Node) *stats.Stats {
        url := fmt.Sprintf("%s/stats", node.Api)
        resp, err := http.Get(url)
        if err != nil {
                log.Printf("Error connecting to %v: %v", node.Api
        }

        if resp.StatusCode != 200 {
                log.Printf("Error retrieving stats from %v: %v",
        }

        defer resp.Body.Close()
        body, _ := ioutil.ReadAll(resp.Body)
        var stats stats.Stats
        json.Unmarshal(body, &stats)
```

```
        return &stats
}
```

The third helper function used by our `Score` method above is the `calculateLoad` function. This function is much simpler than the `calculateCpuUsage` function. It takes two parameters: `usage`, which is of type `float64`, and `capacity`, also a `float64` type. Then, it simply divides `usage` by `capacity` and returns the result.

```
func calculateLoad(usage float64, capacity float64) float64 {
        return usage / capacity
}
```

The final method of our E-PVM scheduler to implement is the `Pick` method. This method is similar to the same method in the round-robin scheduler. It differs only in changing the name of the `lowestScore` variable to `minCost` to reflect the shift to the E-PVM scheduler's focus on *marginal cost*. Otherwise, the method performs the same basic purpose, to select the node with the minimum, or lowest, cost.

Listing 10.22. The E-PVM scheduler's `Pick` method is almost identical to the one in the `RoundRobin` scheduler.

```
func (e *Epvm) Pick(scores map[string]float64, candidates []*node
        minCost := 0.00
        var bestNode *node.Node
        for idx, node := range candidates {
                if idx == 0 {
                        minCost = scores[node.Name]
                        bestNode = node
            continue
                }

                if scores[node.Name] < minCost {
                        minCost = scores[node.Name]
                        bestNode = node
                }
        }
        return bestNode
}
```

With the implementation of the `Pick` method, we have completed the implementation of our second scheduler. This scheduler, like the round-robin

scheduler, implements the `Scheduler` interface. As a result, we can use either scheduler in our manager. Before we change our `main.go` program to use this new scheduler, however, let's take a minor detour and take care of some unfinished business.

10.10 Completing the Node implementation

Earlier, we implemented a helper function named `getNodeStats`. This function takes a variable `node`, which is a pointer to a `node.Node` type. As the name of the function suggests, it communicates with the node by making a `GET` call to the node's `/stats` endpoint. It then returns the resulting stats from the node as a pointer to a `stats.Stats` type. This function is part of the scheduler, so it's awkward to have it handling the lower level details of calling the node's `/stats` endpoint, checking the response, and decoding the response from JSON.

Let's factor this code out of the scheduler and put it where it really belongs, in the `Node` type. We implemented the `Node` type back in chapter 2, so let's review what it looks like since it has been a while since we've seen it.

The `Node` type is pretty straightforward, as we can see in listing 10.23. Its fields hold the values that represent various attributes of our physical or virtual machine that's performing the role of the worker.

Listing 10.23. The `Node` type we defined in chapter 2.

```
type Node struct {
        Name            string
        Ip              string
        Cores           int
        Memory          int
        MemoryAllocated int
        Disk            int
        DiskAllocated   int
        Role            string
        TaskCount       int
}
```

Let's remove the `getNodeStats` function from our `scheduler.go` file and add

it to the `node.go` file in the `node/` package directory. As part of this moving process, let's also change the name to `GetStats`.

Listing 10.24. We rename the `getNodeStats` helper function to `GetStats` and make it a method of the `node.Node` type.

```
func (n *Node) GetStats() (*stats.Stats, error) {
        var resp *http.Response
        var err error

        url := fmt.Sprintf("%s/stats", n.Api)
        resp, err = utils.HTTPWithRetry(http.Get, url)
        if err != nil {
                msg := fmt.Sprintf("Unable to connect to %v. Perm
                log.Println(msg)
                return nil, errors.New(msg)
        }

        if resp.StatusCode != 200 {
                msg := fmt.Sprintf("Error retrieving stats from %
                log.Println(msg)
                return nil, errors.New(msg)
        }

        defer resp.Body.Close()
        body, _ := ioutil.ReadAll(resp.Body)
        var stats stats.Stats
        err = json.Unmarshal(body, &stats)
        if err != nil {
                msg := fmt.Sprintf("error decoding message while
                log.Println(msg)
                return nil, errors.New(msg)
        }

        n.Memory = int64(stats.MemTotalKb())
        n.Disk = int64(stats.DiskTotal())

        n.Stats = stats

        return &n.Stats, nil
}
```

With the `GetStats` method now implemented on the `Node` type, we can remove the old `getNodeStats` helper function from `scheduler.go`. And, finally, we can update the `calculateCpuUsage` helper function to use the

`node.GetStats` method. In addition to using the `node.GetStats` method, let's also change the function signature to return a pointer to a `float64` and an `error`. The changed function looks like this:

```
func calculateCpuUsage(node *node.Node) (*float64, error) {
        //stat1 := getNodeStats(node)
        stat1, err := node.GetStats()
        if err != nil {
                return nil, err
        }
        time.Sleep(3 * time.Second)
        //stat2 := getNodeStats(node)
        stat2, err := node.GetStats()
        if err != nil {
                return nil, err
        }

    // unchanged code

        return &cpuPercentUsage, nil
```

The `GetStats` helper calls a node's worker API, so we need to expose it on the worker `Api` type. This change is simple and can be seen below.

```
func (a *Api) initRouter() {
    // previous code unchanged

        a.Router.Route("/stats", func(r chi.Router) {
                r.Get("/", a.GetStatsHandler)
        })
}
```

With the above changes, we have completed our detour. Now, let's return to our fancy new `E-PVM` scheduler and take it for a spin!

10.11 Using the E-PVM scheduler

At this point, we have all the work completed on our shiny new scheduler interface. We have two types of schedulers we can use in our manager: the `round-robin` scheduler, and the `E-PVM` scheduler. We have already made most of the necessary changes to use the scheduler interface, but we have a few minor tweaks to make that will allow us to easily switch between the

`RoundRobin` and `Epvm` schedulers.

The first change to make involves adding a new `case` to the `switch` statement in the manager's `New` function. The new case adds support for the `Epvm` scheduler.

Listing 10.25. We add a new case to the `switch` statement to support the new "epvm" scheduler.

```
switch schedulerType {
case "roundrobin": #1
        s = &scheduler.RoundRobin{Name: "roundrobin"}
case "epvm" #2
        s = &scheduler.Epvm{Name: "epvm"}
default:
    return nil, fmt.Errorf("scheduler type must be either 'round-
}
```

The second and final change happens in our `main.go` program. If you recall, we previously created a new instance of the manager with the "roundrobin" scheduler using the `New` function.

```
func main() {
        m := manager.New(workers, "roundrobin")
}
```

Now, however, we want to create an instance of the manager that will use the `Epvm` scheduler. To do this, we can simply change the string in the call to `New` from `roundrobin` to `epvm`.

```
func main() {
        m := manager.New(workers, "epvm")
}
```

That's it! We can now run our main program and it will use the `Epvm` scheduler instead of the `RoundRobin` scheduler. Give it a try!

10.12 Summary

- The scheduling problem exists all around us, from home chores to seating customers in a restaurant.

- Scheduling does not have a one-size-fits-all solution. There are multiple solutions, and each one makes tradeoffs based on what we are trying to achieve. It can be as simple as using a round-robin algorithm to select each node in turn. Or, it can be as complex as devising a method to calculate a score for each node based on some set of data, for example the current CPU load and memory usage of each node.
- For the purposes of scheduling tasks in an orchestration system, we can generalize the process to three functions: selecting candidate nodes, which involves reducing the number of possible nodes based on some selection criteria (e.g. does the node have enough disk space to pull the task's container image?); scoring the set of candidate nodes; and, finally, picking the "best" candidate node.
- We can use these three functions to create a general framework to allow us to implement multiple schedulers. In Go, the `interface` is what allows us to create this framework.
- In this chapter, we started three workers, in contrast to a single one in past chapters. Using three workers allowed us to see a more realistic example of how the scheduling process works. However, it's not the same as a more real-world scenario of using multiple physical or virtual machines.

11 Implementing persistent storage for tasks

This chapter covers

- Describing the purpose of a datastore in an orchestration system
- Defining the requirements for our persistent datastore
- Defining the `Store` interface
- Introducing BoltDB
- Implementing the persistent datastore using the `Store` interface
- Discussing the special concerns that exist for the manager's datastore

The fundamental unit of our orchestration system is the `task`. Up until now, we have been keeping track of this fundamental unit by storing it in Go's built-in `map` type. Both our worker and our manager store their respective tasks in a map. This strategy has served us well, but you may have noticed a major problem with it: any time we restart the worker or the manager, they lose all their tasks. The reason they lose their tasks is that Go's built-in map is an in-memory data structure and is not persisted to disk.

Similar to how we revisited our earlier decisions around scheduling tasks, we're now going to return to our decision about how we store them. We're going to talk briefly about the purpose of a datastore in an orchestration system, and then we're going to start the process of replacing our previous in-memory map datastore with a persistent one.

11.1 The storage problem

Why do we need to store the tasks in our orchestration system? While we haven't talked much at all about the issue, the storage of tasks is crucial to a working orchestration system. Storing tasks in some kind of datastore enables the higher level functionality of our system:

- it enables the system to keep track of each task's current state
- it enables the system to make informed decisions about scheduling
- it enables the system to help tasks recover from failures

As we mentioned earlier, our current implementation uses Go's built-in map type, which means we're storing tasks in memory. If we stop the manager, then start it back up because, say, we made a code change, the manager loses the state of all its tasks. We then have no way to recover the system as a whole. For example, if we start our system with three workers and a manager, restarting the manager means we can't gracefully stop running tasks by calling the manager's API, i.e. we can't call `curl -X DELETE http:localhost:5555/tasks/1234567890`. The manager no longer has any knowledge of that task.

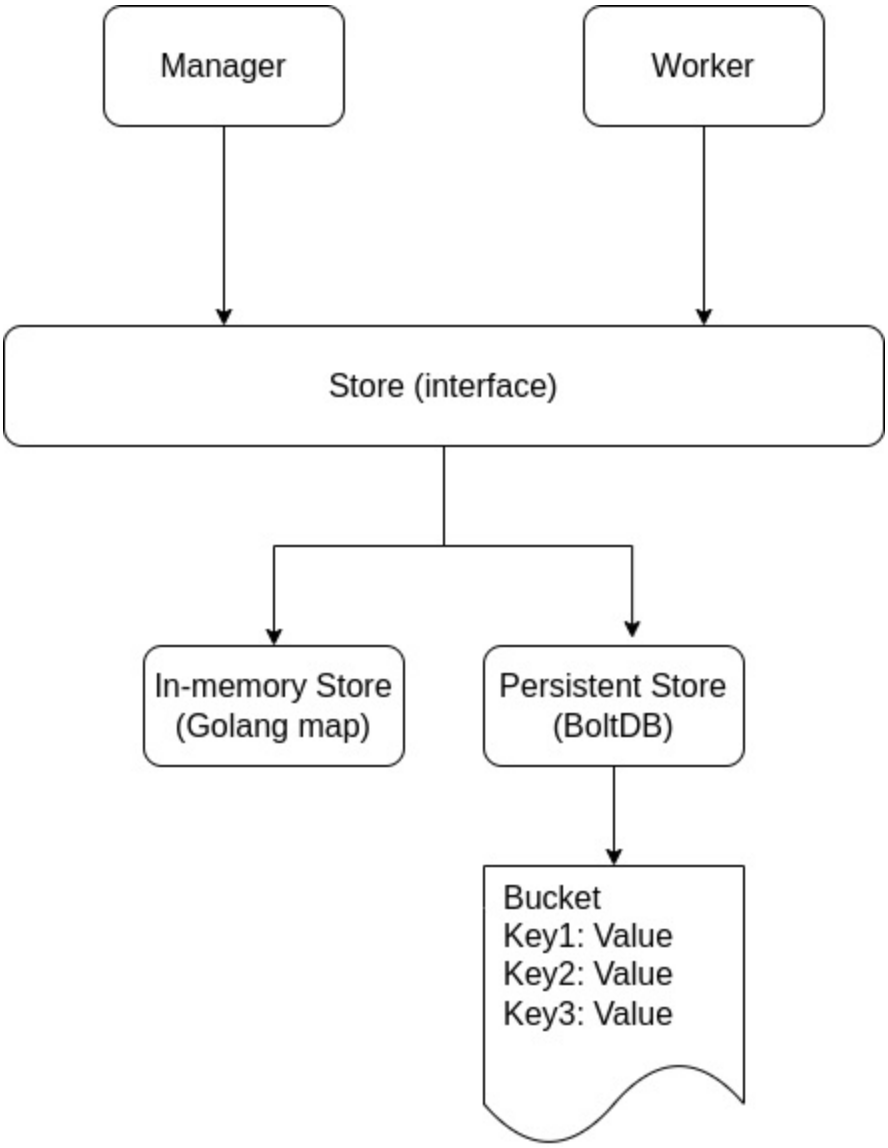
At a basic level, the solution to the above problem is to replace the in-memory map with a persistent datastore. Such a solution will write task state to disk, thus enabling the manager and worker to be restarted without any loss of state.

11.2 The `Store` interface

Before jumping directly to a persistent storage solution, let's follow the same process we used in the last chapter. Remember, we didn't just jump straight to the E-PVM scheduler. Instead, we started by creating the `Scheduler` interface, and then we adapted the existing round-robin scheduler to the interface.

The mental model of our store interface looks like that in figure 11.1. At the top of the model we have the `Manager` and `Worker`, each using the same `Store` interface. That interface is abstract, but as we can see it sits on top of two concrete implementations: an `In-memory Store` and a `Persistent Store`.

Figure 11.1. The mental model of our store interface.



If we think about the operations we have been using to store and retrieve tasks and task events, we can identify four methods to create an interface. Those four methods are:

- Put(key string, value interface{})
- Get(key)
- List()
- Count()

Note

You might be wondering about why the above list doesn't include a `Remove` or `Delete` method. Theoretically, the datastore serves as a historical record of the tasks in an orchestration system. So, it doesn't make sense to provide a method to remove history. In practice, however, it could be useful to provide such a method. For example, over time, an orchestration system would build up a datastore containing tens of thousands, if not hundreds of thousands or more. If the datstore supports the "Remove" operation, it could be used to perform maintenance on the datastore itself.

The `Put` method, as its name suggests, puts an item, identified by a `key`, into the store. Until now, we have been interacting with our store by saving tasks and tasks events directly in a map. For example, in the manager's `SendWork` method, we can see several examples of interacting directly with the `TaskDb` and `EventDb` stores. In the first example below, we pop a task event off the manager's `Pending` queue, convert it to the `task.TaskEvent` type, then store a pointer to the `task.TaskEvent` in the `EventDb` using the task event's id as the key. In the second example, we extract the task from the task event, then store a pointer to it in the `TaskDb` using the task's id as the key.

Listing 11.1. Examples of how we have been using Go's built-in map to store tasks and events.

```
// example # 1
e := m.Pending.Dequeue()
te := e.(task.TaskEvent)
m.EventDb[te.ID] = &te

// example # 2
t := te.Task

// code hidden for the sake of brevity

t.State = task.Scheduled

m.TaskDb[t.ID] = &t
```

There is nothing technically wrong with how we've implemented the task and task event stores up to now. It was quick and easy. More importantly, it just worked. One downside to this implementation, however, is that it is dependent on the underlying data structure underpinning the store. In this case, the manager must know how to put and retrieve items from Go's built-

in map. In other words, we have tightly coupled the manager to the built-in map type. We cannot easily change out this implementation of the store for some other implementation. For example, what if we wanted to use SQLite, a popular SQL-based embedded datastore?

To make it easier for us to use different implementations of a datastore, let's create the `Store` interface seen in listing X.Y. The interface includes the four methods we listed previously, `Put`, `Get`, `List`, and `Count`.

Listing 11.2. The `Store` interface provides the methods that an implementation must meet.

```
type Store interface {
        Put(key string, value interface{}) error
        Get(key string) (interface{}, error)
        List() (interface{}, error)
        Count() (int, error)
}
```

One thing to note in our `Store` interface is that we have declared several values in the method signatures as being of type `interface{}`. The *empty interface*, as this is called, means that the value can be of any type. For example, the `Put` method takes a `key` that is a string and a `value` that is an empty interface, or any type. This means the `Put` method can accept a `value` that is a `task.Task`, or a `task.TaskEvent`, or some other type.

With the `Store` interface defined, let's move on and implement an in-memory store that can replace our existing one.

11.3 Implementing an in-memory store for tasks

We're going to start with an implementation of the task store, and then we'll move on to the task events store. Both the manager and worker use task stores, but only the manager uses an event store. The new implementations of the task and event stores will both wrap Go's built-in map type. By wrapping the built-in map type, we can remove the manager's coupling to the underlying data structure. Instead of needing to understand the mechanics of a map, the manager will simply call the methods of the `Store` interface, and the implementation of the interface will handle all the lower level details of

how to interact with the underlying data structure.

For our purposes, we're going to implement separate types for the task and event stores. We could create a generate store that is able to operate on both tasks and events, but that is more complex and would involve additional concepts that are beyond the scope of this book.

The first implementation is the `InMemoryTaskStore`. We start by defining a struct and giving it a single field called `Db`. Not surprisingly, this field is of type `map[string]*task.Task`, the same as the current implementation. Next, let's define a helper function that will return an instance of the `InMemoryTaskStore`. We'll call this helper function `NewInMemoryTaskStore`, and it takes no arguments and returns a pointer to an `InMemoryTaskStore` that has it's `Db` field initialized to an empty map of type `map[string]*task.Task`.

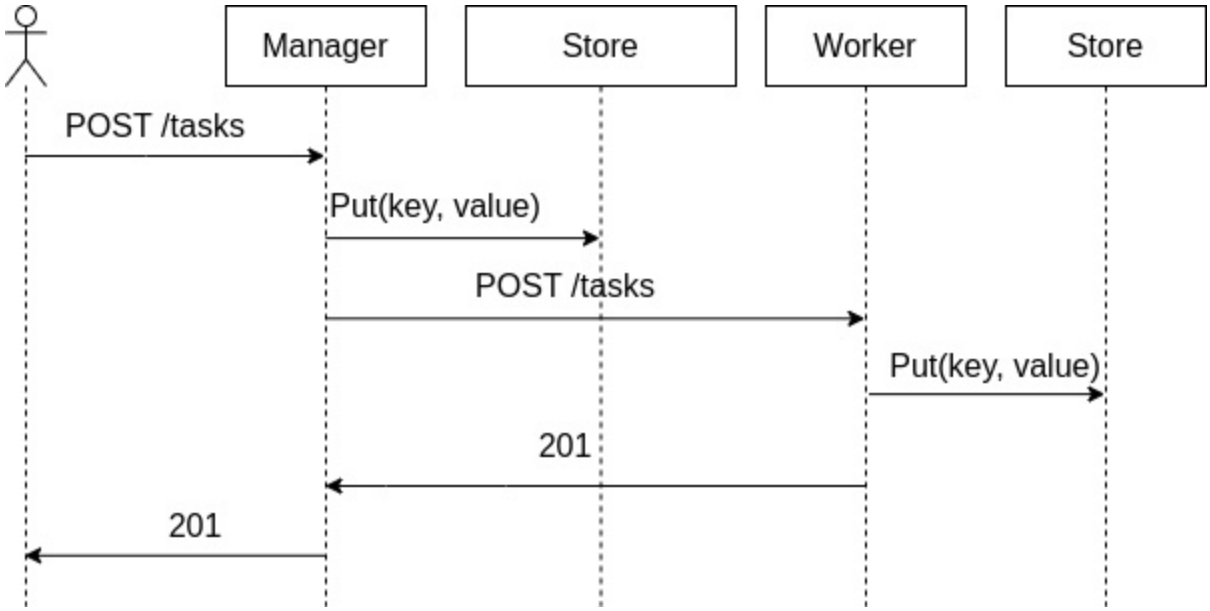
Listing 11.3. The `InMemoryTaskStore` provides a wrapper around Go's built-in map type for the purpose of storing tasks.

```
type InMemoryTaskStore struct {
        Db map[string]*task.Task
}

func NewInMemoryTaskStore() *InMemoryTaskStore {
        return &InMemoryTaskStore{
                Db: make(map[string]*task.Task),
        }
}
```

Let's move on and implement the `Put` method. The sequence diagram in figure 11.2 shows how the `Put` method will be used. When a user calls the manager's API to start a task (`POST /tasks`), the manager will call the `Put` method to store the task in its own datastore. Then, the manager sends the task to the worker by calling the worker's API. The worker, in turn, calls the `Put` method to store the task in its datastore.

Figure 11.2. Sequence diagram illustrating how the manager and worker save tasks to their respective datastores using the `Put` method.



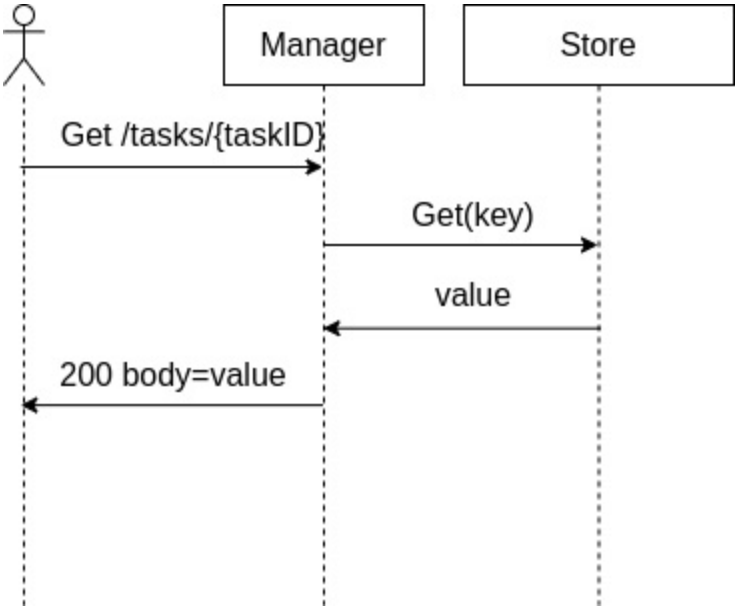
The implementation of the `Put` method, seen in listing X.Y, is fairly straightforward. The method takes two arguments: a `key` that is of type string, and a `value` that is of the empty interface type. In the body of the method, we first attempt to convert the `value` to a concrete type using a *type assertion*. What we are doing is asserting that `value` is not nil and that the value in `value` is a pointer to a `task.Task`. We also capture a boolean named `ok`, which tells us if the assertion was successful. If the assertion was not successful, we return an error, otherwise we store the task `t` in the map.

Listing 11.4. The `Put` method uses the *comma, ok* idiom to verify the type conversion is successful before storing it in the map.

```
func (i *InMemoryTaskStore) Put(key string, value interface{}) er
        t, ok := value.(*task.Task)
        if !ok {
                return fmt.Errorf("value %v is not a task.Task ty
        }
        i.Db[key] = t
        return nil
}
```

Next, we'll implement the `Get` method. The sequence diagram in figure 11.3 shows how the `Get` method will be used. When a user calls the manager's API to get a task (`Get /tasks/{taskID}`), the manager will call the `Get` method to retrieve the task from its datastore and return it.

Figure 11.3. Sequence diagram illustrating how the manager retrieves tasks from its datastore using the `Get` method.



The implementation of the `Get` method takes a `key` of type string and returns an empty interface and potentially an error. We start by looking for the `key` in the store's `Db`. Notice that we're using the *comma ok* idiom here, too. If the `key` exists in `Db`, then `t` will be contain the task identified by the `key`, and we return it. If it does not exist, then `t` will be nil and `ok` will be false and we return an error.

Listing 11.5. The `Get` method uses the *comma, ok* idiom to check if the key exists in the map.

```
func (i *InMemoryTaskStore) Get(key string) (interface{}, error)
        t, ok := i.Db[key]
        if !ok {
                return nil, fmt.Errorf("task with key %s does not
        }

        return t, nil
}
```

The next method we'll implement is the `List` method. Unlike the `Get` method above, which returns a single task, this method returns all the tasks in the store. As you can see below in listing X.Y, we start by creating a variable called `tasks` as a slice of pointers to `task.Task`. This slice will hold all the

tasks in the store. Then, we range over the map in the `Db` field, append each task to the `tasks` slice. Once we've ranged over all the tasks and appended them to the slice, we return it.

Listing 11.6. The `List` method builds up a slice of tasks by ranging over the map. This method always returns `nil` for the `error` value. This is necessary in order to conform to the contract specified by the `Store` interface.

```
func (i *InMemoryTaskStore) List() (interface{}, error) {
        var tasks []*task.Task
        for _, t := range i.Db {
                tasks = append(tasks, t)
        }
        return tasks, nil
}
```

The final method in our task store is `Count`. As its name implies, this method returns the number of tasks contained in the store's `Db` field. Because `Db` is a map, we can get the count of items using the built-in `len` function.

Listing 11.7. The `Count` method can easily get the count of tasks using Go's built-in `len` function.

```
func (i *InMemoryTaskStore) Count() (int, error) {
        return len(i.Db), nil
}
```

Now that we've implemented an in-memory version of the task store, let's move on and do the same thing for task events.

11.4 Implementing an in-memory store for task events

The store for the task events will be identical to the one for tasks. The obvious difference will be the task events store will operate on the `task.TaskEvent` type and not `task.Task`. Because the differences are minor, we won't go into the details.

Listing 11.8. The `InMemoryTaskEventStore` is identical to the `InMemoryTaskStore`, with the exception of operating on `task.TaskEvent` types instead of `task.Task`.

```
type InMemoryTaskEventStore struct {
        Db map[string]*task.TaskEvent
}

func NewInMemoryTaskEventStore() *InMemoryTaskEventStore {
        return &InMemoryTaskEventStore{
                Db: make(map[string]*task.TaskEvent),
        }
}

func (i *InMemoryTaskEventStore) Put(key string, value interface{
        e, ok := value.(task.TaskEvent)
        if !ok {
                return fmt.Errorf("value %v is not a task.TaskEve
        }
        i.Db[key] = &e
        return nil
}

func (i *InMemoryTaskEventStore) Get(key string) (interface{}, er
        e, ok := i.Db[key]
        if !ok {
                return nil, fmt.Errorf("task event with key %s do
        }

        return e, nil
}

func (i *InMemoryTaskEventStore) List() (interface{}, error) {
        var events []*task.TaskEvent
        for _, e := range i.Db {
                events = append(events, e)
        }
        return events, nil
}

func (i *InMemoryTaskEventStore) Count() (int, error) {
        return len(i.Db), nil
}
```

11.5 Refactoring the manager to use the new in-memory stores

At this point, we have defined an interface that will allow us to store tasks and events. We have also implemented two concrete types of our store

interface, both of which wrap Go's built-in map type and remove the need for the manager and worker to interact with it directly. So, let's make some changes to the manager and worker so they can make use of our new code.

Starting with the manager, we need to update the `TaskDb` and `EventDb` fields on the `Manager` struct. Instead of these fields being of type `map[uuid.UUID]*task.Task` and `map[uuid.UUID]*task.TaskEvent`, let's change them both to be of type `store.Store`. With this change, our manager can now use any kind of store that implements the `store.Store` interface.

```
type Manager struct {
    // fields omitted for convenience
        TaskDb        store.Store
        EventDb       store.Store
    // fields omitted
```

Changing the `TaskDb` and `EventDb` fields to an interface type should look familiar to you. If you remember in chapter 10, we did something similar when we introduced the `Scheduler` field, which was of type `scheduler.Scheduler`, also an interface. That change allowed us to configure the manager to use different types of schedulers, and now we have configured it to use different types of stores.

Next, let's modify the `New` function in the manager package. In the last chapter, we updated it so it accepted a `schedulerType` in addition to a slice of `workers`. Now, let's add another argument to the `New` function, one called `dbType`. The new signature will look like the following:

```
func New(workers []string, schedulerType string, dbType string) *
```

There are several changes to the body of the `New` function that we'll need to make next. The first of these changes is to remove the initialization of the `taskDb` and `eventDb` variables using the `make()` built-in function. Just delete those two lines for now. We're going to do something slightly different here in a bit.

Now, we want to change how we're returning from the function. We are currently returning a pointer to the `Manager` type like this:

```
return &Manager{ ... }
```

Instead of returning a pointer using what's called a *struct_literal*, let's assign it to a variable named `m`. It will look like this:

Listing 11.9. Assigning a struct literal to the `m` variable instead of returning it.

```
m := Manager{
        Pending:       *queue.New(),
        Workers:       workers,
        WorkerTaskMap: workerTaskMap,
        TaskWorkerMap: taskWorkerMap,
        WorkerNodes:   nodes,
        Scheduler:     s,
}
```

At this point, we have an instance of our `Manager` type, but it does not have any stores for tasks and events. We're going to use the `dbType` variable we added to the `New` function's signature as part of a `switch` statement to allow us to set up different types of datastores based on the value of `dbType`. Because we've only implemented in-memory stores, we're going to start by only supporting the case where the value of `dbType` is `memory`. In this case, we call the `NewInMemoryTaskStore` function to create an instance of our in-memory task store, and we call the `NewInMemoryTaskEventStore` function to create an instance of our in-memory event store.

All that's left to do now is to assign the value of the `ts` variable to the manager's `TaskDb` field, and assign the `es` value to the manager's `EventDb` field. Then, return a pointer to the manager.

```
var ts store.Store #1
var es store.Store
switch dbType { #2
case "memory": #3
        ts = store.NewInMemoryTaskStore()
        es = store.NewInMemoryTaskEventStore()
}

m.TaskDb = ts #4
m.EventDb = es #5
return &m #6
```

Now we're ready for the substantive changes! We need to change the manager's methods so it interacts with the datastore using the methods of our

new `Store` interface instead of operating directly on the map structures. The first method we'll work on is `updateTasks`.

All of the changes we need to make in the `updateTasks` method occur inside the for loop that ranges over a slice of pointers to `task.Task` types. The first change to make involves replacing the block of code that checks if an individual task reported by a worker exists in the manager's task store. The current code uses the *comma ok* idiom to perform this check. We'll replace this block with a call to the store interface's `Get` method, and checking the `err` value to indicate the task doesn't exist in the manager's store.

We can see this change below in listing X.Y. The existing code is commented out, and its replacement follows just afterward. Now, our updated code is calling the `Get` method on the manager's `TaskDb` store, passing it the `ID` of the task as a string. If the task exists in the manager's store, it will be assigned to the `result` variable, and if there is an error it will be assigned to the `err` variable. Next, we perform the usual error checking, and if there is an error, we log it and move on to the next task using the `continue` statement. Finally, we use a type assertion to convert the `result`, which is of type `interface{}` to the concrete `task.Task` type (actually a pointer to a `task.Task`). If the type assertion fails, then we log a message and continue on to the next task.

```
for _, t := range tasks {

    // previous code omitted for convenience

    // existing code to be replaced
        // _, ok := m.TaskDb[t.ID]

        // if !ok {

        //      log.Printf("[manager] Task with ID %s not found\n

        //      continue

        // }

    result, err := m.TaskDb.Get(t.ID.String())
        if err != nil {
                log.Printf("[manager] %s", err)
                        continue
        }
```

```
        taskPersisted, ok := result.(*task.Task)
        if !ok {
                        log.Printf("cannot convert result %v to t
                        continue
        }
```

The last set of changes to make in the `updateTasks` method involve replacing the remaining direct operations on the existing map structure with calls to the appropriate methods of the `Store` interface. The existing code can be seen in listing X.Y. Here, we are modifying a task by directly changing its fields in the map.

```
if m.TaskDb[t.ID].State != t.State {

        m.TaskDb[t.ID].State = t.State

}

m.TaskDb[t.ID].StartTime = t.StartTime

m.TaskDb[t.ID].FinishTime = t.FinishTime

m.TaskDb[t.ID].ContainerID = t.ContainerID

m.TaskDb[t.ID].HostPorts = t.HostPorts

}
```

We want to replace the above code with that below in listing X.Y. Since we have already retrieved the task from the manager's task store, and we've converted it from an empty interface type to a pointer to the concrete `task.Task` type, we can simply update the necessary fields on the `taskPersisted` variable. Then, we finish up by calling the store's `Put` method to save the updated task.

Listing 11.10. The `Store` interface allows us to clean up our interaction with the `taskPersisted` type.

```
if taskPersisted.State != t.State {
                taskPersisted.State = t.State
}

taskPersisted.StartTime = t.StartTime
```

```
taskPersisted.FinishTime = t.FinishTime
taskPersisted.ContainerID = t.ContainerID
taskPersisted.HostPorts = t.HostPorts

m.TaskDb.Put(taskPersisted.ID.String(), taskPersisted)
```

The `doHealthChecks` method is the next method we need to update. It uses a for loop to range over all the tasks in the manager's task store. Up to now, this method has been ranging directly over the map of tasks. Instead of doing that, let's implement a helper method that will use the `Store` interface's `List` method, build a slice of tasks, and return us that slice. We can see this new helper method, named `GetTasks` in listing X.Y.

Listing 11.11. The `GetTasks` method calls the `List` method and converts the result from an empty interface to a slice of pointers to `task.Type`.

```
func (m *Manager) GetTasks() []*task.Task {
        taskList, err := m.TaskDb.List() #1
        if err != nil { #2
                log.Printf("error getting list of tasks: %v", err
                return nil
        }

        return taskList.([]*task.Task) #3
}
```

With the `GetTasks` helper method implemented, let's turn our attention back to the `doHealthChecks` method, where we'll use it. The first step is to call `GetTasks` and store the result in a variable called `tasks`. Now that we have a slice of tasks, we just need to change the for loop to range over `tasks` instead of directly over the map.

```
// for _, t := range m.TaskDb {
//        // code omitted
// }

tasks := m.GetTasks()
for _, t := range tasks {
        // omitted code
}
```

The next method that needs updating is the `restartTask` method. This one is easy. It currently has a single interaction with the map built-in, so all we need

to do is replace it with a call to the store's `Put` method. So, it's just a matter of replacing the first line below with the second.

```
// m.TaskDb[t.ID] = t
m.TaskDb.Put(t.ID.String(), t)
```

The final method to update is the `SendWork` method. Despite being a long method that encompasses a multi-step process to send tasks to workers, we have only a few updates to make here. The first update involves our first interaction with the new `EventDb` store. We want to change from interacting directly with the old task events map to using the new `EventDb` store. Early on in the `SendWork` method, we pop an event off the manager's `Pending` queue, and we convert it to `task.TaskEvent` type. Now, we want to call the `Put` method on the events store, passing it the event `ID` as a string and a pointer to the task event `te`. If the call to `Put` returns an error, we log it and return.

Listing 11.12. The first change to the `SendWork` method, which involves using the new `Put` method instead of operating directly on the map.

```
e := m.Pending.Dequeue()
te := e.(task.TaskEvent)
err := m.EventDb.Put(te.ID.String(), &te)
if err != nil {
                log.Printf("error attempting to store task event
                return
}
```

The second update involves this method's use of the tasks store. In listing X.Y below, we can see the existing code where we are again interacting directly with the `TaskDb` map. Since this code is getting a task from the map, we want to convert the code to use the store interface's `Get` method like we've done previously.

Listing 11.13. Existing code that gets a task from the store by operating directly on the map.

```
taskWorker, ok := m.TaskWorkerMap[te.Task.ID]
if ok {
                persistedTask := m.TaskDb[te.Task.ID]
                if te.State == task.Completed && task.ValidStateT
                        m.stopTask(taskWorker, te.Task.ID.String(
```

```
                        return
                }
}
```

To change the above code to use our new `Get` method on the store interface, we need to rearrange our code a little.

```
result, err := m.TaskDb.Get(te.Task.ID.String()) #1
if err != nil { #2
                log.Printf("unable to schedule task: %s", err)
                return
}

persistedTask, ok := result.(*task.Task) #3
if !ok { #4
                log.Printf("unable to convert task to task.Task t
                return
}
```

The final update to the `SendWork` method, and our final update to the manager, involves another change to use the task store's `Put` method instead of inserting a task directly in a map.

```
t.State = task.Scheduled
// m.TaskDb[t.ID] = &t
m.TaskDb.Put(t.ID.String(), &t)
```

11.6 Refactoring the worker

At this point our manager is using the new `Store` interface. Our worker, however, is not. It's still operating directly on the built-in map type. So, let's perform the same refactoring on the worker so it, too, uses the `Store` interface.

Like the manager, the first order of business is to change the worker's `Db` type from a `map[uuid.UUID]*task.Task` to the `store.Store` interface type. By doing so, it can use any type of store that implements the `store.Store` interface.

```
type Worker struct {
    // fields omitted
        Db    store.Store
```

```
    // fields omitted
}
```

Next, we need to update the `New` helper function in the `worker` package. Let's update its function signature to take another argument. This new argument is named `taskDbType` and is a string. We then create a variable `s` that is of the new `store.Store` type. Now, we use a switch statement on the `taskDbType` argument and assign the result of the `NewInMemoryTaskStore` function call to the variable `s`. Finally, we assign `s` to the worker's `Db` field. We can then return a pointer to the worker `w`, which will include the store interface.

Listing 11.14. The `New` helper function now creates an instance of the `InMemoryTaskStore`.

```
func New(name string, taskDbType string) *Worker {
        w := Worker{
                Name:  name,
                Queue: *queue.New(),
        }

        var s store.Store
        var err error
        switch taskDbType {
        case "memory":
                s = store.NewInMemoryTaskStore()
        }
        w.Db = s
        return &w
}
```

With the change to the `New` helper function, let's move on to the worker's methods. The first to modify is the `GetTasks` method. Remember, this method was previously operating directly on the worker's `Db` map field. Since we've moved the logic that operates directly on the underlying store (in this case a built-in map), we want `GetTasks` to use the store interface instead. We want to replace the body of `GetTasks` with the simplified version seen here in list X.Y.

Listing 11.15. The body of the `GetTasks` method calls the store interface's `List` method.

```
func (w *Worker) GetTasks() []*task.Task {
        taskList, err := w.Db.List()
        if err != nil {
```

```
            log.Printf("error getting list of tasks: %v", err
            return nil
        }

        return taskList.([]*task.Task)
```

Next, we need to modify the `runTask` method. Like all of the previous code, it too has been operating directly on the `Db` map. The beginning steps of this method are the following:

1. Pop a task off the worker's `Queue`
2. Convert the task from an interface to a `task.Task` type
3. Get the task from the `Db` map
4. If the task doesn't exist, then create it

Notice steps 3 and 4. The process attempts to get the task from the map by looking it up using the task's `ID`. This operation, however, doesn't return an error if the task isn't in the map. Our `InMemoryTaskStore` implements the `Store` interface's `Get` method, which does returns an `error`. That error could be due to any number of factors. It could be because the task simply didn't exist in the store, or because there was some issue interacting with the underlying store itself. So, if we were to use the same order of operations when we switch to using the new `Store` interface, we'd have a problem. If the call to the store's `Get` method returns an error, how do we distinguish if it's because the task didn't exist or if there was an error with the underlying store itself? In the former case, we want to create the task; in the latter case, we want to return an error.

As we've done in the past, our solution is making a tradeoff. We're going to switch the order of operations so that we call the store's `Put` method first, which will effectively overwrite the task if it exists. If the call to `Put` does not return an error, then we call the store's `Get` method to retrieve the task from the store.

Listing 11.16. The `runTask` method changes the order of operations to account for our `Store` interface including errors in return values.

```
func (w *Worker) runTask() task.DockerResult {
    // previous code omitted
```

```
        err := w.Db.Put(taskQueued.ID.String(), &taskQueued)
        if err != nil {
                msg := fmt.Errorf("error storing task %s: %v", ta
                log.Println(msg)
                return task.DockerResult{Error: msg}
        }

        result, err := w.Db.Get(taskQueued.ID.String())
        if err != nil {
                msg := fmt.Errorf("error getting task %s from dat
                log.Println(msg)
                return task.DockerResult{Error: msg}
        }

    // code omitted
```

The `StartTask` method is the next one that requires changes. It performs two operations on the task store. Each one is updating the state of the task and storing the updated task in the `Db`. In these cases, we can simply swap the direct operation on the map with a call to the new `Put` method, as seen in listing X.Y.

Listing 11.17. Using the `Put` method instead of directly operating on a map.

```
func (w *Worker) StartTask(t task.Task) task.DockerResult {
        config := task.NewConfig(&t)
        d := task.NewDocker(config)
        result := d.Run()
        if result.Error != nil {
                log.Printf("Err running task %v: %v\n", t.ID, res
                t.State = task.Failed
                w.Db.Put(t.ID.String(), &t)
                return result
        }

        t.ContainerID = result.ContainerId
        t.State = task.Running
        w.Db.Put(t.ID.String(), &t)
```

Next, the `StopTask` method operates on the task store just once. Similar to the `StartTask` method, it is updating the state of the task and saving it to the map. Again, we can simply swap out the direct interaction with the map and replace it with a call to the store's `Put` method.

```
func (w *Worker) StopTask(t task.Task) task.DockerResult {
        config := task.NewConfig(&t)
        d := task.NewDocker(config)

        stopResult := d.Stop(t.ContainerID)
        if stopResult.Error != nil {
                log.Printf("%v\n", stopResult.Error)
        }
        removeResult := d.Remove(t.ContainerID)
        if removeResult.Error != nil {
                log.Printf("%v\n", removeResult.Error)
        }

        t.FinishTime = time.Now().UTC()
        t.State = task.Completed
        w.Db.Put(t.ID.String(), &t)
        log.Printf("Stopped and removed container %v for task %v\

        return removeResult
}
```

Finally, the `updateTasks` method operates on the task store four times. The first operation is a for loop that ranges over the worker's `Db` map. Because Go supports iterating over a map, we were able to loop over the store directly. Go doesn't support iterating over function call—it only returns once.

```
func (w *Worker) updateTasks() {
        // for each task in the worker's datastore:
        // 1. call InspectTask method
        // 2. verify task is in running state
        // 3. if task is not in running state, or not running at
        tasks, err := w.Db.List() #1
        if err != nil {
                log.Printf("error getting list of tasks: %v", err
                return
        }
        for _, t := range tasks.([]*task.Task) { #2
                if t.State == task.Running {
                        resp := w.InspectTask(*t)
                        if resp.Error != nil {
                                fmt.Printf("ERROR: %v", resp.Erro
                        }

                        if resp.Container == nil {
                                log.Printf("No container for runn
                                t.State = task.Failed
```

```
                        w.Db.Put(t.ID.String(), t) #3
                    }

                    if resp.Container.State.Status == "exited
                        log.Printf("Container for task %s
                        t.State = task.Failed
                        w.Db.Put(t.ID.String(), t) #4
                    }

                    // task is running, update exposed ports
                    t.HostPorts = resp.Container.NetworkSetti
                    w.Db.Put(t.ID.String(), t) #5
                }
        }
}
```

11.7 Putting it all together

At this point, we're almost ready to spin up our manager and workers and have them use the new `Store` interface. All that's needed are a few minor tweaks to our `main.go` program.

The first tweak to make involves how we create our workers. If you recall, we had been creating them by assigning a struct literal to a variable.

```
w1 := worker.Worker{
                Queue: *queue.New(),
                Db:    store.NewInMemoryTaskStore(),
}
```

Now, however, we can simplify this part of our code by using the `New` helper function in the worker package. So we'll replace the three lines above with a single call to the `New` function.

```
w1 := worker.New("worker-1", "memory")
w2 := worker.New("worker-2", "memory")
w3 := worker.New("worker-3", "memory")
```

The second tweak involves how we're creating the manager. We already had a `New` helper function that we were using. We now need to add an argument to our call to `New` that specifies what type of datastore the manager should use.

```
m := manager.New(workers, "epvm", "memory")
```

With these changes, we can now run our main program and see what we get.

```
$ CUBE_WORKER_HOST=localhost CUBE_WORKER_PORT=5556 CUBE_MANAGER_H
Starting Cube worker
Starting Cube manager
2023/03/04 16:19:38 No tasks to process currently.
2023/03/04 16:19:38 Sleeping for 10 seconds.
2023/03/04 16:19:38 Checking status of tasks
2023/03/04 16:19:38 Task updates completed
2023/03/04 16:19:38 Sleeping for 15 seconds
2023/03/04 16:19:38 Checking status of tasks
2023/03/04 16:19:38 Task updates completed
2023/03/04 16:19:38 Sleeping for 15 seconds
2023/03/04 16:19:38 Processing any tasks in the queue
2023/03/04 16:19:38 No work in the queue
2023/03/04 16:19:38 Sleeping for 10 seconds
2023/03/04 16:19:38 No tasks to process currently.
2023/03/04 16:19:38 Sleeping for 10 seconds.
2023/03/04 16:19:38 Checking for task updates from workers
2023/03/04 16:19:38 Checking worker localhost:5556 for task updat
```

As you can see, not much has changed. The workers and manager start up as expected and do their respective jobs.

Let’s send a task to the manager.

```
curl -X POST localhost:5555/tasks -d @task1.json

2023/03/04 16:19:42 Add event {a7aa1d44-08f6-443e-9378-f588431101

2023/03/04 16:19:48 Pulled {a7aa1d44-08f6-443e-9378-f5884311019e

2023/03/04 16:19:57 [manager] selected worker localhost:5556 for
2023/03/04 16:19:57 [worker] Added task bb1d59ef-9fc1-4e4b-a44d-d
2023/03/04 16:19:57 [manager] received response from worker

[worker] Found task in queue: {bb1d59ef-9fc1-4e4b-a44d-db571eeed2

2023/03/04 21:19:59 Listening on http://localhost:7777
```

It works! So, we have successfully refactored the manager and worker to use an interface representing a datastore instead of operating directly on the datastore itself. There is still one problem. If we stop and restart either the

manager or the worker, they will forget about any tasks they have previously seen.

We can solve this problem by implementing a persistent datastore.

11.8 Introducing BoltDB

In moving from an in-memory datastore to a persistent one, there are some high-level questions we have to ask ourselves. First, do we need a *server-based* datastore? A server-based datastore is like PostgreSQL, MySQL, Cassandra, Mongo, or any other datastore that runs as its own process. For our purposes, a server-based datastore is overkill. It would be another process we'd have to start and then manage, and most server-based systems can get complex quickly.

Instead, we're going to choose an *embedded* datastore. This is called an *embedded* datastore because it uses a library that you "embed" directly in your application.

The second question involves the data model that we want to use. The most popular data model is the relational model, which is what systems like PostgreSQL and MySQL use. There is even an embedded relational datastore, SQLite. While such datastores are popular and robust, they also require the use of the *Structured Query Language*, or SQL, to insert and query data. SQL datastores are highly structured and require strict schemas defining tables and columns.

Another data model that has become popular in the last decade is the key-value datastore, sometimes also referred to as NoSQL. Popular opensource key/value datastores include Cassandra and Redis.

If you recall our in-memory datastore using Go's built-in `map` type, it had a simple interface: we *put* data into the datastore, and we *got* data out of it. The main mechanism by which we put or got tasks into this datastore was the `key`, in our case a `UUID`. Our tasks are the *values* to which the *keys* refer.

Because we are already using a key-value datastore, it makes sense to pick a

persistent datastore that uses the same paradigm. And, to keep this as simple as possible, we're going to use an embedded library called BoltDB (github.com/boltdb/bolt). As mentioned in BoltDB's README, it is "a pure Go key/value store" and the "goal of the project is to provide a simple, fast, and reliable database for projects that don't require a full database server such as Postgres or MySQL".

To use the BoltDB library, we will need to install it. From your project directory, install the library using the command below:

```
$ go get github.com/boltdb/bolt/...
```

As we did with the in-memory version of of the task and event datastores, we are now going to implement persistent versions of each store.

11.9 Implementing a persistent task store

The first persistent store we will implement is the `TaskStore`. It will implement the `Store` interface, the same as the in-memory stores do. The only difference will be in the implementation details.

The first thing to do is to create the `TaskStore` struct seen below in listing X.Y. There are several differences from the in-memory version to note. The first is that the `TaskStore` struct uses a different type for its `Db` field. Whereas the `InMemoryTaskStore` used a `map[string]*task.Task` type, here the field is a pointer to the `bolt.DB` type. This type is defined in the BoltDB library. Next, The `TaskStore` struct defines the `DbFile` and `FileMode` fields. BoltDB uses a file on disk to persist data, and the `DbFile` field tells BoltDB the name of the file it will operate on, and the `FileMode` ensures that we have the necessary permissions to read and write to the file. In BoltDB, key/value pairs are stored in collections called "buckets", so the struct's `Bucket` field defines the name of the bucket we want to use for the `TaskStore`.

Listing 11.18. The persistent version of our task store is called `TaskStore`.

```
import (
        // previous imports omitted
```

```
        "github.com/boltdb/bolt"
)

type TaskStore struct {
        Db       *bolt.DB
        DbFile   string
        FileMode os.FileMode
        Bucket   string
}
```

The next thing to do is to create a helper function to create an instance of our persistent datastore. We did the same thing with our in-memory datastores. The `NewTaskStore` helper takes three arguments: a string `file` that provides the name of the file we want to use, the `mode` of the file as an `os.FileMode` type, and a string that provides the name of the bucket in which we want to store our tasks.

Listing 11.19. The `NewTaskStore` helper function creates an instance of our persistent task datastore.

```
func NewTaskStore(file string, mode os.FileMode, bucket string) (
        db, err := bolt.Open(file, mode, nil) #1
        if err != nil {
                return nil, fmt.Errorf("unable to open %v", file)
        }
        t := TaskStore{ #2
                DbFile:   file,
                FileMode: mode,
                Db:       db,
                Bucket:   bucket,
        }

        err = t.CreateBucket() #3
        if err != nil {
                log.Printf("bucket already exists, will use it in
        }

        return &t, nil #4

}
```

With the `TaskStore` struct defined and our helper function created, let's turn our attention to the methods of the `TaskStore`. In addition to the methods defined by the `Store` interface--`Put`, `Get`, `List`, and `Count`--we're going to

define a `Close` method. Why do we need such a method? Remember, our persistent datastore is writing its data to a file on disk. Moreover, in the `NewTaskStore` helper function, we called the `Open` function to open the file. The `Close` method will close the file when we're done with it.

```
func (t *TaskStore) Close() {
        t.Db.Close()
}
```

The first method of the `Store` interface to implement is `Count`. As with the in-memory stores, the persistent version will return the number of tasks in the datastore. Unlike the in-memory versions, this one is a little more involved.

Similar to relational datastores like PostgreSQL or MySQL, BoltDB supports transactions. Per the BoltDB README, each transaction in BoltDB "has a consistent view of the data as it existed when the transaction started." BoltDB supports three types of transactions:

- Read-write
- Read-only
- Batch read-write

For our purposes, we will only be using the first two.

Since our `Count` method needs to get the number of tasks, we can use a read-only transaction. Unlike the in-memory stores, where we were able to use Go's built-in `len` method on Go's map type, here we have to iterate over all the keys in the bucket to construct our count. BoltDB provides the `ForEach` method to simplify this process.

To perform a read-only transaction we use Bolt's `View` function. This method takes a function, which itself takes an argument of a pointer to a `bolt.Tx` type and returns an error. This is the mechanism that provides the transaction. Inside the transaction, we identify the bucket on which we want to operate, then iterate over each key in that bucket and increment the `taskCount` value. After iterating over all the keys, we check for any errors before finally return the `taskCount`.

```
func (t *TaskStore) Count() (int, error) {
```

```
        taskCount := 0
        err := t.Db.View(func(tx *bolt.Tx) error { #1
                b := tx.Bucket([]byte("tasks")) #2
                b.ForEach(func(k, v []byte) error { #3
                        taskCount++ #4
                        return nil
                })
                return nil
        })
        if err != nil { #5
                return -1, err
        }

        return taskCount, nil #6
}
```

Let's move on and implement another method that is specific to the persistent store and is thus not part of the `Store` interface. The method is the `CreateBucket` method seen in listing X.Y. It takes no arguments and returns an error. In this method, we create the bucket that will hold all of our tasks as key/value pairs. Because we are creating a bucket, we need to use a read-write transaction, and we do this with the `Update` function. Using `Update` works similar to `View`. It takes a function that takes a pointer to a `bolt.Tx` type and returns an error. We then call the `CreateBucket` function and pass in the name of the bucket to create. We then check for any errors.

Listing 11.20. Our `CreateBucket` method is a wrapper around a function of the same name in the BoltDB library.

```
func (t *TaskStore) CreateBucket() error {
        return t.Db.Update(func(tx *bolt.Tx) error {
                _, err := tx.CreateBucket([]byte(t.Bucket))
                if err != nil {
                        return fmt.Errorf("create bucket %s: %s",
                }
                return nil
        })

}
```

Now, let's return to the methods of the `Store` interface and implement the second of these methods, `Put`. The signature of this `Put` method is the same as for the in-memory versions, so there is nothing new here. What is new is how

we store the key/value pair. As we did in the `CreateBucket` method above, we will use the `Update` function to get a read-write transaction. We identify the bucket where we will store the key/value pair using the `tx.Bucket` function, passing it the name as a string. In order to store the value in the BoltDB bucket, we have to convert the value to a slice of bytes. We do this by calling the `Marshal` function from the `json` package, and passing it the `value` converted to a pointer to the `task.Task` type. Once the value has been converted to a slice of bytes, we call the `Put` function on the bucket, passing it the key and value (both as slices of bytes).

Listing 11.21. TODO

```
func (t *TaskStore) Put(key string, value interface{}) error {
        return t.Db.Update(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(t.Bucket))

                buf, err := json.Marshal(value.(*task.Task))
                if err != nil {
                        return err
                }

                err = b.Put([]byte(key), buf)
                if err != nil {
                        return err
                }
                return nil
        })
}
```

The third method of the `Store` interface to implement is the `Get` method. It takes a string, which is the key we want to retrieve. It returns an `interface{}`, which is the task we wanted, and an error. We start by defining the variable `task` of type `task.Task`. Next, we use a read-only transaction by way of the `View` function, which we've seen above. Again, we identify the bucket on which we want to operate. Then, we look up the task using the `Get` function, passing it the `key` as a slice of bytes. Notice that we do not check the `Get` call for any errors. The reason for this is that the BoltDB library guarantees `Get` will work unless there is a system failure (e.g. the datastore file is deleted from disk). If there is no task in the bucket for the `key`, then `Get` returns `nil`. Once we have the task, we have to decode it from a slice of bytes back into a `task.Type`, which we do using the `Unmarshal` function from the

`json` package. Finally, we do some error checking, then return a pointer to the task.

```
func (t *TaskStore) Get(key string) (interface{}, error) {
        var task task.Task
        err := t.Db.View(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(t.Bucket))
                t := b.Get([]byte(key))
                if t == nil {
                        return fmt.Errorf("task %v not found", ke
                }
                err := json.Unmarshal(t, &task)
                if err != nil {
                        return err
                }
                return nil
        })
        if err != nil {
                return nil, err
        }
        return &task, nil

}
```

The fourth and final method to implement is the `List` method. Like `Get` above, `List` uses a read-only transaction. Instead of getting a single task, however, it iterates over all the tasks in the bucket and creates a slice of tasks. In this, it is similar to the `Count` method.

```
func (t *TaskStore) List() (interface{}, error) {
        var tasks []*task.Task
        err := t.Db.View(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(t.Bucket))
                b.ForEach(func(k, v []byte) error {
                        var task task.Task
                        err := json.Unmarshal(v, &task)
                        if err != nil {
                                return err
                        }
                        tasks = append(tasks, &task)
                        return nil
                })
                return nil
        })
        if err != nil {
                return nil, err
```

```
        }

        return tasks, nil
}
```

11.10 Implementing a persistent task event store

The persistent store for task events will look almost the same as the one for tasks. The obvious differences will be in naming. Our struct is named `EventStore` instead of `TaskStore`, and it will operate on the `task.TaskEvent` type instead of `task.Task`.

The `EventStore` struct and `NewEventStore` helper function should look familiar. There isn't much to discuss here.

```
type EventStore struct {
        DbFile   string
        FileMode os.FileMode
        Db       *bolt.DB
        Bucket   string
}

func NewEventStore(file string, mode os.FileMode, bucket string)
        db, err := bolt.Open(file, mode, nil)
        if err != nil {
                return nil, fmt.Errorf("unable to open %v", file)
        }
        e := EventStore{
                DbFile:   file,
                FileMode: mode,
                Db:       db,
                Bucket:   bucket,
        }

        err = e.CreateBucket()
        if err != nil {
                log.Printf("bucket already exists, will use it in
        }

        return &e, nil
}
```

Likewise, the `Close` and `CreateBucket` methods on the `EventStore` type

should also look familiar. The former is closing the datastore file opened in `NewEventStore`, and the latter is creating a bucket to store events.

```
func (e *EventStore) Close() {
        e.Db.Close()
}

func (e *EventStore) CreateBucket() error {
        return e.Db.Update(func(tx *bolt.Tx) error {
                _, err := tx.CreateBucket([]byte(e.Bucket))
                if err != nil {
                        return fmt.Errorf("create bucket %s: %s",
                }
                return nil
        })
}
```

The `Count` method counts the number of events in the persistent store, returning the count.

```
func (e *EventStore) Count() (int, error) {
        eventCount := 0
        err := e.Db.View(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(e.Bucket))
                b.ForEach(func(k, v []byte) error {
                        eventCount++
                        return nil
                })
                return nil
        })
        if err != nil {
                return -1, err
        }

        return eventCount, nil
}
```

The `Put` and `Get` methods are also identical to their counterparts in the `TaskStore` type. `Put` takes a key and a value, writes it to the datstore and returns any errors. `Get` takes a key, looks it up in the datastore and returns the value, if found.

```
func (e *EventStore) Put(key string, value interface{}) error {
        return e.Db.Update(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(e.Bucket))
```

```
                buf, err := json.Marshal(value.(*task.TaskEvent))
                if err != nil {
                        return err
                }

                err = b.Put([]byte(key), buf)
                if err != nil {
                        log.Printf("unable to save item %s", key)
                        return err
                }
                return nil
        })
}

func (e *EventStore) Get(key string) (interface{}, error) {
        var event task.TaskEvent
        err := e.Db.View(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(e.Bucket))
                t := b.Get([]byte(key))
                if t == nil {
                        return fmt.Errorf("event %v not found", k
                }
                err := json.Unmarshal(t, &event)
                if err != nil {
                        return err
                }
                return nil
        })

        if err != nil {
                return nil, err
        }
        return &event, nil
}
```

And, last but not least, the `List` method builds a list of all the events in the datastore and returns it.

```
func (e *EventStore) List() (interface{}, error) {
        var events []*task.TaskEvent
        err := e.Db.View(func(tx *bolt.Tx) error {
                b := tx.Bucket([]byte(e.Bucket))
                b.ForEach(func(k, v []byte) error {
                        var event task.TaskEvent
                        err := json.Unmarshal(v, &event)
                        if err != nil {
```

```
                                return err
                        }
                        events = append(events, &event)
                        return nil
                })
                return nil
        })
        if err != nil {
                return nil, err
        }

        return events, nil
}
```

With the persistent versions of our task and event stores implemented, we can change our main program to use them instead of their in-memory counterparts.

11.11 Switching out the in-memory stores for permanent ones

We need to make a couple of minor changes in our manager and worker code in order to use the new persistent stores. Both changes involve adding cases for creating persistent datastores in the `New` helper functions in the manager and worker packages.

Let's start with the manager. We need to add the `persistent` case to our `switch` statement, seen in listing X.Y. Thus, when a caller of the `New` function passes in `persistent` as the value of `dbType`, we call the `NewTaskStore` and `NewEventStore` functions instead of their in-memory equivalents. Notice that each function takes three arguments: the name of the file to use for the datastore, the filemode of the file, and the name of the bucket that will store the key/value pairs.

Listing 11.22. Adding the "persistent" case to the `New` helper function in the manager package. The `0600` in the function calls represent the filemode argument, which means only the owner of the file can read and write it.

```
switch dbType {
case "memory":
```

```
        ts = store.NewInMemoryTaskStore()
        es = store.NewInMemoryTaskEventStore()
case "persistent":
        ts, err = store.NewTaskStore("tasks.db", 0600, "tasks")
        es, err = store.NewEventStore("events.db", 0600, "events"
}
```

The changes in the worker's `New` function are similar. We add a `persistent` case, which calls the `NewTaskStore` helper function. We're starting three workers, so we use the `filename` variable to create a unique filename for each worker. Because the worker only operates on tasks, there is no need to set up an event store.

Listing 11.23. Adding the "persistent" case to the `New` helper function in the worker package.

```
case "persistent":
        filename := fmt.Sprintf("%s_tasks.db", name)
        s, err = store.NewTaskStore(filename, 0600, "tasks")
}
```

At this point, changing our main program to use the new persistent store is just a matter of changing four lines of existing code. In all four lines, seen below, we simply change the string `memory` to `persistent`.

```
//w1 := worker.New("worker-1", "memory")
w1 := worker.New("worker-1", "persistent")

//w2 := worker.New("worker-2", "memory")
w2 := worker.New("worker-2", "persistent")

// w3 := worker.New("worker-3", "memory")
w3 := worker.New("worker-3", "persistent")

//m := manager.New(workers, "epvm", "memory")
m := manager.New(workers, "epvm", "persistent")
```

With these changes, start up the main program and perform the same operations that we performed earlier in the chapter. You should notice that everything looks the same from the outside as it did when we used the in-memory stores. The only difference is that you will now see several files with the `.db` extension in the working directory. These are the files BoltDB is using to persist the systems tasks and events. The files you should see are:

- tasks.db
- worker-1_tasks.db
- worker-2_tasks.db
- worker-3_tasks.db

11.12 Summary

- Storing the orchestrator's tasks and events in persistent datastores allows the system to keep track of task and event state, to make informed decisions about scheduling, and to help recover from failures.
- The `store.Store` interface enables us to swap out datastore implementations based on our needs. For example, while doing development work, we can use an in-memory store, while we use a persistent store for production.
- While we adapted our old stores that were based on Go's built-in map type to the new `store.Store` interface, these in-memory implementations suffer the same problem, that is the manager and worker will still lose their tasks when they restart.
- With the `store.Store` interface and a concrete implementation, we made changes to the manager and worker to remove their operating directly on the datastore. For example, instead of operating on a map of `map[uuid.UUID]*task.Task`, we changed them to operate on the `store.Store` interface. In doing this, we decoupled the manager and worker from the underlying datastore implementation: they no longer needed to know the internal workings of the specific datastore; they only needed to know how to call the methods of the interface while all the technical details were handled by an implementation.
- The BoltDB library provides an embedded key/value datastore on top of which we built our `TaskStore` and `EventStore` stores. These datastores persist their data to files on disk, thus allowing the manager and worker to gracefully restart without losing their tasks.
- Once we created the `store.Store` interface and two implementations (one in-memory, one persistent), we could switch between the implementations by simply passing a string of either "memory" or "persistent" to the `New` helper functions.