Deploying Rails with Docker, Kubernetes and ECS

Pablo Acuña

Apress®

Deploying Rails with Docker, Kubernetes and ECS

Pablo Acuña

Apress®

Deploying Rails with Docker, Kubernetes and ECS

Pablo Acuña
Providencia, Chile

ISBN-13 (pbk): 978-1-4842-2414-4 ISBN-13 (electronic): 978-1-4842-2415-1
DOI 10.1007/978-1-4842-2415-1

Library of Congress Control Number: 2016961433

Managing Director: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Todd Green, Celestin Suresh John, Nikhil Karkal, Robert Hutchinson, James Markham, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Nancy Chen
Copy Editor: Lori Jacobs
Compositor: SPi Global
Indexer: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

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

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Printed on acid-free paper

To Mamá & Papá.

Contents at a Glance

Contents

About the Author

Pablo Acuña is a software engineer from the south of Chile. He specializes in Ruby on Rails applications and containerized systems. He holds a computer engineering degree from Universidad Técnica Federico Santa Maria in Valparaíso, Chile and currently works as a Lead Developer at ArchDaily—the world's most visited architecture website.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in Security, Web/Mobile development, and Cloud and IT Architecture. His true information technology (IT) passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Massimo's technical skills include: Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies and he is the coauthor of *Pro Android Games* (Apress, 2015).

Acknowledgments

I'd like to thank all who have taken part in this journey with me—especially Becky, for always being there for me and supporting all of my ambitious endeavors. I would also like to thank ArchDaily and its Amazing Dev team for giving me the opportunity and to explore these new technologies. Finally, to my family: without you none of this would have been possible. Thank you.

Introduction

There are a lot of different container orchestration platforms out there. Every one of them has a different set of features that sets them apart. Choosing one of these platforms can be a complex task. You have to dig into the specifics of each one of these platforms to see if the concepts and the mechanisms behind the framework make sense to you. It's almost impossible to judge them without trying them for a while. Sadly, not many people have the time to try every one of the current alternatives.

On top of that problem, people spend a lot of time figuring out how to deploy a specific application using containers. Every web application framework out there requires different configuration, different initialization steps, and different tasks to run updates. Even though there are some practices in the world of containers that could be considered standard, the differences between the applications always create a new problem that has to be solved in a custom way. For this reason, it's critical to choose an orchestration platform that adds support for all these requirements and provides the tools to create a smooth workflow when running updates.

In this book you'll see the main steps to deploy a specific type of applications, a Ruby on Rails application. If you are a Rails developer, you should be pretty familiar with the typical steps to run a Rails application in development and production. If you don't use Rails, this book should also give you valuable information on how to create templates to run tasks and web applications using Docker with Kubernetes and ECS for deployment.

After you choose the right platform to deploy your application, you have to deal with the issue of running the whole infrastructure. You have to create and manage a cluster and be able to add or remove nodes when necessary. In this book you'll see how Kubernetes and ECS deal with these tasks in a very transparent way. You can launch an entire cluster with a couple of commands without being an expert on topics such as networking or DNS (Domain Name Systems).

Once you have your application up and running, you can build a Continuous Integration (CI) pipeline around it. The whole idea is to be able to run deployments very often without having downtime and to make sure that we are not pushing bad code to our version control system. Automated testing plays a big part in this structure. We should be able to run our test suite before sending our code to production servers. For that we also have several alternatives out there. Here, we are going to be using Jenkins as our CI server. Jenkins is a classical choice when it comes to CI. It's highly configurable and it plays nicely with the tools we need, such as Docker, Kubernetes, and AWS (Amazon Web Services).

About the Book

You should take the information given in this book as a set of ideas on how to implement your deployment pipelines. Your application is going to be different; it might have more dependencies or it might require more build steps. But at the end, as a long you understand how to apply the basic concepts behind containers, and especially Kubernetes and ECS, you'll be able to create pipelines for any application. In this book you'll find example scripts, templates, shell commands, and a lot of other information that should help you to build your system.

We are going to use a simple Rails application as an example. We are going to scaffold a resource so we have migrations to play with and also create records along the way to test our end points. This scaffold will also generate a couple of tests, with which we are going to add a build step that runs our tests in the CI pipeline. We want to reject deployments with code that breaks our tests.

I recommend that you read Chapters 3 and 4, so you can compare Kubernetes and ECS and make a smart decision on which platform you'll use. Also, some of the issues we discuss throughout this book are explored in more depth in each of those chapters.

CHAPTER 1

Development

In this chapter we are going to create a new Rails application from scratch using the latest version. We are going to build an API-only application so we can test it easily. Also, the idea of using containers to deploy applications makes more sense when we have small applications. That's because, in this kind of architecture, we'll be pulling and pushing images to a Docker registry all the time, and it's better to have lightweight images if we want to have faster deploys.

If you are an active member of the development community, you probably have noticed that the concept of services is becoming more and more important in web development. That's why we are going to focus on these types of applications, and an API (application programming interface) is a good representation of a small service that we can successfully deploy using containers.

Dependencies

The only dependencies we need are Docker and Docker Compose. A nice thing about using containers is that we don't have to pollute our system with a lot of dependencies. In our case we are going to build a Rails application with a PostgreSQL database, but we won't install any of those.

The Docker installation will change depending on your system. You can find the proper way to install it on your operating system in the official documentation (`https://docs.docker.com/engine/installation/`).

Once you have Docker installed, you can check the version with `docker version`. The version I'll be using is `1.12.2`.

The other dependency we need, is Docker Compose. You can find the installation instructions on the web site (`https://docs.docker.com/compose/install/`). Again, this installation will be different depending on your operating system (OS). After that, check your version with `docker-compose version`. I'll be using version `1.8.1`.

Creating the Application

Let's create a new API-only (`http://edgeguides.rubyonrails.org/api_app.html`) Rails application using PostgreSQL (`www.postgresql.org/`) as our database. This API will be in charge of managing articles with a small set of fields. We just want something we can

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1_1

play with. Since this book is based on containers, I'll show you a couple of tricks to create a new Rails application without having any dependencies other than Docker installed on your machine.

We are going to use the official Rails image (https://hub.docker.com/_/rails/) from DockerHub to create this new application. With that image we'll run a container and pass an entry point with the `rails new` command along with some options.

Let's generate the app with the following command:

```
$ docker run -it --rm --user "$(id -u):$(id -g)" -v "$PWD":/usr/src/app -w /
usr/src/app \
rails rails new --skip-bundle --api --database postgresql webapp
```

This command will create a Rails app named `webapp` in our current directory. This command might look complicated but it's actually pretty simple. We are running a command using the `rails` image. We are mounting a volume from the container folder which generates the application into our current folder. The `-user` option is necessary for permission reasons. Then we are overriding the entry point of the command with a custom command to generate the skeleton.

As you can see, in the entry point we are passing the `--api` flag, so Rails generates an API-only application, and also the `--database` flag, so the drivers for our database are pre-configured. The `--skip-bundle` flag is going to tell Rails not to run `bundle install`. This is necessary because some gems like `pg` for the PostgreSQL database sometimes require us to install dependencies on our machine. Here we are going to create a new Rails app but without its dependencies. That's OK, since we want to install the dependencies in the actual container instead of on our machine.

Since we created the application without running `bundle install`, we don't have a `Gemfile.lock` file to maintain the gem versions. Our container approach requires us to have that file. Thus, we are going to use another trick to generate it. This time we are going to mount the recently created application in a ruby container and run `bundle install` from within the container. The command will generate a `Gemfile.lock` file which will also be in our directory product of the volume mount:

```
$ cd webapp
$ docker run --rm -v "$PWD":/usr/src/app -w /usr/src/app ruby:2.3 bundle
install
```

This time we run a container from the `ruby:2.3` image, again mounting the current folder into the container and then running `bundle install` as the main command.

And that's it. Now we have our `Gemfile` and `Gemfile.lock` and the application is ready to be dockerized.

Since we don't have any dependencies installed on our machine, we cannot test this application in the typical way by running `rails s`. Remember, we don't even have Rails installed. So we'll have to wait a little bit until we can run the `webapp` within a Docker container.

Dockerizing Rails

With all the Docker images available these days, it's pretty easy to dockerize any type of application. In our case we only going to need something for the Rails app and something for PostgreSQL. For Rails you have practically the same options you have when you don't use containers for deployments (Unicorn, Puma, Passenger, etc). And for PostgreSQL we can use the official image from DockerHub.

If you explore DockerHub, you can find almost any image you need. And if you don't, you can always use a base image, and build a new one by adding the dependencies you need on top of it.

For Ruby/Rails I strongly recommend that you use the phusion passenger-docker (https://github.com/phusion/passenger-docker) image. It has a lot of features and a very nice documentation. You can have a solid production application server out of the box with that image, so we will use it throughout this book. Keep in mind that if you choose another application server like `unicorn` or `puma`, the setup is going to be quite different.

First we are going to create a development environment with Docker Compose.

This environment is going to be just for development. For production we want to use a more robust tool with cluster management support like Kubernetes or ECS.

Even though we don't want to use Docker Compose for production, it's a very realistic scenario to see that our containers are interacting with each other and if you have a working environment with Docker Compose, it's going to be much easier to have a working production environment. That's the beauty of containers.

The application needs to be built using the phusion passenger-docker as the base image. So we need a Dockerfile. This Dockerfile is also going to add the application source code along with other configuration files to the container.

The typical configuration files that you need for this image are the Nginx Virtual Host for your application and a file to declare the environmental variables you may need to pass to the application. First, let's create a file for the virtual host configuration. You can create all of these files in the root of the `webapp` application:

```
$ touch webapp.conf
```

And add the following to that file:

```
server {
    listen 80;
    server_name _;
    root /home/app/webapp/public;

    passenger_enabled on;
    passenger_user app;

    passenger_ruby /usr/bin/ruby2.3;
}
```

Here we have a very simple Virtual Host configuration for Nginx and Passenger. This file will be added to the available hosts of the server during the container build.

Now let's add a file for the environmental variables. We'll put some just so we know how to declare them if we want to use them later:

```
$ touch rails-env.conf
```

And add the following:

```
env SECRET_KEY_BASE;
env DATABASE_URL;
env DATABASE_PASSWORD;
```

We won't be using those variables during development so don't worry about them for now. You just need to know that every environmental variable that you need in your application should be declared here so they are preserved and forwarded to your web application.

Now we can create the Dockerfile that's going to add those files to the entire application.

I've added comments to every instruction so you will know what's happening at every step of the build. If you still have doubts about an instruction, just go to the official repository web site (https://github.com/phusion/passenger-docker) of the image and take a look at the documentation; it's really good.

```
$ touch Dockerfile
```

And the content:

```
FROM phusion/passenger-ruby23:0.9.19

# Set correct environment variables.
ENV HOME /root

# Use baseimage-docker's init process.
CMD ["/sbin/my_init"]

# Additional packages: we are adding the netcat package so we can
# make pings to the database service
RUN apt-get update && apt-get install -y -o Dpkg::Options::="--force-
confold" netcat

# Enable Nginx and Passenger
RUN rm -f /etc/service/nginx/down

# Add virtual host entry for the application. Make sure
# the file is in the correct path
RUN rm /etc/nginx/sites-enabled/default
ADD webapp.conf /etc/nginx/sites-enabled/webapp.conf
```

```
# In case we need some environmental variables in Nginx. Make sure
# the file is in the correct path
ADD rails-env.conf /etc/nginx/main.d/rails-env.conf

# Install gems: it's better to build an independent layer for the gems
# so they are cached during builds unless Gemfile changes WORKDIR /tmp
ADD Gemfile /tmp/
ADD Gemfile.lock /tmp/
RUN bundle install

# Copy application into the container and use right permissions: passenger
# uses the app user for running the application RUN mkdir /home/app/webapp
COPY . /home/app/webapp
RUN usermod -u 1000 app
RUN chown -R app:app /home/app/webapp
WORKDIR /home/app/webapp

# Clean up APT when done.
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

EXPOSE 80
```

This Dockerfile should do the trick for our Rails application. It might seem a little bit complicated at first, but trust me, you'll get used to it after building a couple of Rails services.

The cool thing about this setup, is that you'll be using the same server for development and production, so you'll have a lot less work to do once you ship the application to production. This image in particular uses the `PASSENGER_APP_ENV` variable to set the environment. So, for example, in Rails, that variable also controls the `RAILS_ENV` variable value.

Now that we have the Dockerfile to build our application, we need a Docker Compose file so we can run the database and then test these containers to see if everything works well.

We'll need to declare three services in our Docker Compose file: one for our web application, one for the PostgreSQL database, and one setup container for running initialization commands. This setup container will be run before the actual application, and it's going to migrate our database.

■ **Tip** It's good practice to run tasks in separate containers. In a production scenario you may want to run several containers for your web application, and if you run initialization commands in that same container (such as migrations), you'll have collisions, since all the containers will be running those commands before the main container starts.

Setup Container

As I mentioned previously, we need a mechanism to run initialization commands. In a Rails application, these are typically `rails db:create`, `rails db:migrate`. With the exception of the `rails assets:precompile` command, we can't run those commands in the same container that we're running our web application. That would work well if you're deploying only one container, but suppose you want to scale your application and you want to run ten containers; then every one of those containers will try to execute those commands during startup time and things can get ugly. With that said, it's always better to separate concerns and to use an independent container to run tasks and then just remove it once it finishes.

This setup container has to use the same Dockerfile that we just created since it needs access to the whole application environment. The only part we need to overwrite is the entry point.

Right now we don't have an entry point for our container since we want to use the one the same image is providing for us. But since this setup container doesn't need to run as a web application, we can just overwrite it with our own initialization commands. Let's create this new entry point script in our root path.

```
$ touch setup.sh
```

And add the following:

```
#!/bin/sh

echo "Waiting PostgreSQL to start on 5432..."

while ! nc -z postgres 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

bin/rails db:migrate
```

The whole purpose of this script is to run the possible new migrations our application may have. But, what if the database is not yet available for the command? That would generate an error and the whole run would crash. That's why we use a `while` expression that's going to loop until the connection is alive. We are using the `netcat` package for this and also we're assuming that the PostgreSQL service end point will be reachable by using the `postgres` alias. Finally, when the connection is alive, we can run the latest migrations.

Let's also add the proper execution permissions for this file.

```
$ chmod +x setup.sh
```

This type of script is very typical when you need to orchestrate different services that are constantly being shut down and turned on. So you'd better get used to getting your hands dirty with some bash when you run containers.

Let's create the `docker-compose.yml` file in our root path and add the service

```
$ touch docker-compose.yml
```

And the first service

```
version: '2'
services:
  webapp_setup:
    build: .
    depends_on:
      - postgres
    environment:
      - PASSENGER_APP_ENV=development
    entrypoint: ./setup.sh
```

This service will use the Dockerfile we already have for the build. It has a dependency on a service we haven't declared yet and which we will name `postgres`.

We are also passing an environmental variable for the Rails environment. This is one of the ways of setting the `RAILS_ENV` variable for this image and it's described in the documentation.

Finally, we are overwriting the entry point with the previous bash script. Since this script doesn't do anything after the migration, the container will be exited when it finishes, which is want we want.

Web Application Container

This service will be similar to the setup service. Following are the differences:

- We need to declare a dependency on the setup container so it always runs after it.
- We need to map the port 80 of the container to our host if we want to access our application via HTTP.
- We don't need to override the entry point.

We can express all of that with the following:

```
webapp:
  container_name: webapp
   build: .
  depends_on:
    - postgres
    - webapp_setup
  environment:
    - PASSENGER_APP_ENV=development
  ports:
    - "80:80"
```

As you can see in this service, we are not touching the entry point. The default entry point provided by the base image will be executed and the application will be launched as a web application. Those two dependencies will assure us that this container will be started after the database is ready for accepting connections and the setup container has been executed.

Database Container

The PostgreSQL service will be very simple. We need to pull the `postgres:9.5.3` image from DockerHub and set up a couple of environmental variables like the user, password, and database name:

```
postgres:
   image: postgres:9.5.3
   environment:
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_USER=webapp
      - POSTGRES_DB=webapp_development
```

In this case our user will be `webapp` and we want to use the default database name that Rails set up for us in the database config file.

Since the container will create this database during startup, we don't need to run `rails db:create` in our application initialization script.

Keep in mind that for a daily development workflow, you **want** to mount your source file in the application container. That way you can see your changes immediately inside the container. You can accomplish this with just one line of code.

```
volumes:
  - .:/home/app/webapp
```

That code is going to mount your local source code to the folder where the application lives inside the container.

You also may want to create a data-only container for holding our database data. This way, in case we destroy the postgres container for some reason, our development data would still be there. This pattern is useful for all other kind of stateful applications, like search engines, cache stores, and so on.

We can add this data-only container with the following:

```
postgres_data:
    image: postgres:9.5.3
    volumes:
      - /var/lib/postgresql/data
    command: /bin/true
```

And we can use the volume from the postgres service by adding

```
volumes_from:
      - postgres_data
```

To the service.

The final `docker-compose.yml` file should look as follows:

```
version: '2'
services:
  webapp_setup:
    build: .
    depends_on:
      - postgres
    environment:
      - PASSENGER_APP_ENV=development
    entrypoint: ./setup.sh
  webapp:
    container_name: webapp
    build: .
    depends_on:
      - postgres
      - webapp_setup
    environment:
      - PASSENGER_APP_ENV=development
    ports:
      - "80:80"
    volumes:
      - .:/home/app/webapp
    postgres:
      image: postgres:9.5.3
      environment:
        - POSTGRES_PASSWORD=mysecretpassword
        - POSTGRES_USER=webapp
        - POSTGRES_DB=webapp_development
      volumes_from:
        - postgres_data
    postgres_data:
        image: postgres:9.5.3
        volumes:
          - /var/lib/postgresql/data
        command: /bin/true
```

Build and Run

Let's run the Docker Compose build command to build our web application and setup containers. During this section we'll be using the most common Docker Compose commands. One important thing is that you should always run these commands in the same directory where the `docker-compose.yml` file is.

```
$ docker-compose build
```

Output (truncated):

```
postgres_data uses an image, skipping
postgres uses an image, skipping
Building webapp_setup
Step 1 : FROM phusion/passenger-ruby23:0.9.19 0.9.19:
Pulling from phusion/passenger-ruby23
f069f1d21059: Extracting [===>] 3.146 MB/49.17 MB
ecbeec5633cf: Download complete
ea6f18256d63: Download complete
54bde7b02897: Waiting
a3ed95caeb02: Waiting
ce9e695a6234: Waiting
346026b9659b: Waiting
ffaf5356e027: Waiting
85417a8aee4f: Waiting
...
```

The first time you run this command it may take some time, since the Docker engine has to pull the image, install the dependencies, and install the gems inside the container. If you follow the output of the build, you should see all these steps.

At the end you should see something like the following in the output:

```
    (truncated)
Building webapp
Step 1 : FROM phusion/passenger-ruby23:0.9.19
---> 6841e494987f
Step 2 : ENV HOME /root
---> Using cache
---> e70985107acc
Step 3 : CMD /sbin/my_init
---> Using cache
---> babd8b525225
Step 4 : RUN apt-get update && apt-get install -y -o Dpkg::Options::="--
force-confold" netcat
---> Using cache
---> 74da8c84b454
Step 5 : RUN rm -f /etc/service/nginx/down
---> Using cache
---> 16d62000d878
Step 6 : RUN rm /etc/nginx/sites-enabled/default
---> Using cache
---> 0b6a404fc6cc
Step 7 : ADD webapp.conf /etc/nginx/sites-enabled/webapp.conf
---> Using cache
---> 5d1327265ff2
```

```
Step 8 : ADD rails-env.conf /etc/nginx/main.d/rails-env.conf
---> Using cache
---> 45b636122c30
Step 9 : WORKDIR /tmp
---> Using cache
---> 55878be0e6a5
Step 10 : ADD Gemfile /tmp/
---> Using cache
---> ed9c23c126a3
Step 11 : ADD Gemfile.lock /tmp/
---> Using cache
---> af7aeac5d540
Step 12 : RUN bundle install
---> Using cache
---> 6259538ad852
Step 13 : RUN mkdir /home/app/webapp
---> Using cache
---> eec9872fe976
Step 14 : COPY . /home/app/webapp
---> Using cache
---> 2155626c9eb4
Step 15 : RUN usermod -u 1000 app
---> Using cache
---> 4f0275216dce
Step 16 : RUN chown -R app:app /home/app/webapp
---> Using cache
---> ee6949c8496c
Step 17 : WORKDIR /home/app/webapp
---> Using cache
---> 40d5c1499bc8
Step 18 : RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
---> Using cache
---> 0c39cab889a8 Step 19 : EXPOSE 80
---> Using cache
---> 4d463695a9f1
Successfully built 4d463695a9f1
```

That whole section is for the webapp container. As you can see, the layers were already cached, since they're shared with the setup container, which was built first. That's just for showing you that even though we have to build two containers, which both run `bundle install`, the build is made virtually just once and then the layers are all cached.

Before running `docker-compose up`, which will start the whole environment, we have to configure our database credentials. Remember we are using an alias for the postgres host.

Open the `config/database.yml` file and change the default section to the following:

```
default: &default
  adapter: postgresql
  encoding: unicode
  user: webapp
  password: mysecretpassword
  host: postgres
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
```

Now we can run the Docker Compose up command to pull the PostgreSQL image, run the initialization script, and set up the entire system:

```
$ docker-compose up
```

Following are some interesting parts you can see in the output:

```
Pulling postgres (postgres:9.5.3)...
5c90d4a2d1a8: Extracting [=========================> ] 50.86 MB/51.35 MB
5c90d4a2d1a8: Downloading [=========================> ] 28.25 MB/51.35 MB co\
mplete
c3961b297acc: Download complete
...
```

For the PostgreSQL image pull, also

```
webapp_setup_1  | PostgreSQL started
postgres_1      | LOG: incomplete startup packet
postgres_1      | LOG: database system was shut down at 2016-09-25
                       21:52:32 UTC
postgres_1      | LOG: MultiXact member wraparound protections are now
                       enabled
postgres_1      | LOG: database system is ready to accept connections
postgres_1      | LOG: autovacuum launcher started
webapp_webapp_setup_1 exited with code 0
```

For the setup container in action. We don't have any migrations yet, so the container is exited with no output.

Also, if you look at the line that says

```
webapp_setup_1    | Waiting PostgreSQL to start on 5432...
```

You can see the setup container trying to connect to the PostgreSQL container in order to run the migrations, but the PostgreSQL initialization wasn't ready yet.

Finally you'll see something like the following:

```
webapp         | [ 2016-09-25 21:52:33.0180 29/7f54586a2780 age/Wat/
WatchdogMain.cpp:1291 ]: Start\
ing Passenger watchdog...
webapp         | [ 2016-09-25 21:52:33.0431 32/7f7e4c98f780 age/Cor/
CoreMain.cpp:982 ]: Starting P\
assenger core...
webapp         | [ 2016-09-25 21:52:33.0433 32/7f7e4c98f780 age/Cor/
CoreMain.cpp:235 ]: Passenger \
core running in multi-application mode.
webapp         | [ 2016-09-25 21:52:33.0453 32/7f7e4c98f780 age/Cor/
CoreMain.cpp:732 ]: Passenger \
core online, PID 32
webapp         | [ 2016-09-25 21:52:33.0665 37/7fb989671780 age/Ust/
UstRouterMain.cpp:529 ]: Start\
ing Passenger UstRouter...
webapp         | [ 2016-09-25 21:52:33.0673 37/7fb989671780
nger UstRouter online, PID 37
```

Which indicates that the passenger and Nginx processes are ready and listening.

Let's test the application via cURL. You should replace the localhost address with whatever you're using for running Docker. In my case I'm using Docker for Mac, so my Docker IP (Internet Protocol) is localhost. Also remember not to kill the Docker Compose up process.

Open a different tab and test the root path.

```
$ curl -I localhost
```

Output:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Connection: keep-alive
Status: 200 OK
Cache-Control: max-age=0, private, must-revalidate
ETag: W/"b62d4f67b7b823c017534cd9727752cd"
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff X-Runtime: 0.020162
X-Request-Id: 5df5e6c8-a441-4582-ab32-67bd6b148279
Date: Sun, 25 Sep 2016 21:57:01 GMT
X-Powered-By: Phusion Passenger 5.0.29
Server: nginx/1.10.1 + Phusion Passenger 5.0.29
```

Great! Our application is running correctly on the development environment.

Adding a Rails Resource

We said this was going to be an API for managing articles, so let's create a simple articles resource. For this, we can run a simple command with Docker Compose, which uses our same webapp service but also overrides the entry point with whatever we want.

Open a new terminal tab in your project's root folder and let's scaffold a new resource with the following:

```
$ docker-compose run --rm webapp bin/rails g scaffold articles title:string
body:text
```

Output:

```
Starting webapp_postgres_data_1
Starting webapp_webapp_setup_1
[WARNING] The model name 'articles' was recognized as a plural, using the
singular 'article' instead\
. Override with --force-plural or setup custom inflection rules for this
noun before running the gen\ erator.
        invoke  active_record
        create    db/migrate/20160925220117_create_articles.rb create
                  app/models/article.rb
        invoke    test_unit
        create      test/models/article_test.rb
        create      test/fixtures/articles.yml
        invoke  resource_route
        route     resources :articles
        invoke  scaffold_controller
        create    app/controllers/articles_controller.rb
        invoke    test_unit
        creat     test/controllers/articles_controller_test.rb
```

That's going to create the necessary files for the scaffold, and since we have a volume for this project, we also have those files locally. That's a workflow you have to learn when working with containers. Locally you're only editing files, but all the tasks and executions happen inside the container, so you normally want to run commands with docker-compose and add the `--rm` flag so the container is deleted after, or you may want to keep an open connection inside the container by using `docker exec -it webapp bash`, and run all your commands from there.

The scaffold we just created is actually pretty cool because it detects that our application is API-only, so it generates a controller already adapted for JSON (JavaScript Object Notation) responses.

Let's migrate the database.

```
$ docker-compose run --rm webapp bin/rails db:migrate
```

Output:

```
Starting webapp_postgres_data_1
Starting webapp_webapp_setup_1
== 20160925220117 CreateArticles:
migrating ===================================
-- create_table(:articles)
   -> 0.0621s
== 20160925220117 CreateArticles: migrated (0.0622s)
============================
```

If we want to run the tests Rails created for us, we first have to create the test database

```
$ docker-compose run --rm webapp bash -c "RAILS_ENV=test bin/rails
db:create"
```

And then run the tests.

```
$ docker-compose run --rm webapp bash -c "RAILS_ENV=test bin/rake"
```

Output:

```
Starting webapp_postgres_data_1
Starting webapp_webapp_setup_1
Run options: --seed 4172

# Running:

.....

Finished in 0.954391s, 5.2389 runs/s, 7.3345 assertions/s.

5 runs, 7 assertions, 0 failures, 0 errors, 0 skips
```

Let's test the end point for creating articles. We are going to use cURL for interacting with our API. Run the following command, but first, make sure the `docker-compose up` command is still running:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"my first
article","body":"Lorem ips\ um dolor sit amet, consectetur adipiscing
elit..."}' http://localhost/articles
```

Output:

```
{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit..\
.","created_at":"2016-09-25T22:10:19.407Z","updated_at":"2016-09-
25T22:10:19.407Z"}%
```

That means the article was indeed created in our database. Just in case you don't believe me let's run a Rails console and inspect a little bit.

```
$ docker-compose exec webapp bin/rails c


Loading development environment (Rails 5.0.0.1)
2.3.1 :001 > Article.first
  Article Load (0.6ms) SELECT "articles".* FROM "articles" ORDER BY
"articles"."id" ASC LIMIT $1 \
[["LIMIT", 1]]
  => #<Article id: 1, title: "my first article", body: "Lorem ipsum dolor
sit amet, consectetur adipi\
scing...", created_at: "2016-09-25 22:10:19", updated_at: "2016-09-25
22:10:19">
```

We can use the `exec` command to open extra processes inside our containers. In this case I'm running the `rails c` command which will run the console in that container and keep the interactive mode so we can run commands. As you can see, we have the record we just created via cURL, so we are pretty sure our setup is working properly.

■ **Tip** Use `docker-compose up -d` to run your environment in detached mode so you can keep using that same terminal window to interact with your container.

Log Issues

A very important and complex topic with containers is Logging. The Docker logging mechanism works by inspecting the `STDOUT` connection. That's a problem for a Rails application, since all the logging is piped to a file in the log's directory. Fortunately for us, we just need to change one line in our source code to send the logs to the `STDOUT` connection.

Open the `config/application.rb` file and add the following configuration:

```
config.logger = Logger.new(STDOUT)
```

And that's it. Now, for example, you should be able to see the application logs with the **docker- compose logs** tool. To apply this change, let's stop the `docker-compose up` command with `C-c (Control + c)` and rebuild the project:

```
$ # kill the process with C-c
```

Output:

```
Killing webapp ... done
Killing webapp_postgres_1 ... done

$ docker-compose build
```

We are going to run `docker-compose up` in detached mode and follow the logs with the logs command:

```
$ docker-compose up -d && docker-compose logs -f
```

Now, open a new tab and again run a command using cURL:

```
$ curl -I localhost
```

If you see the logs in the other window, you should see the Rails logger in action.

```
webapp        | App 443 stdout: I, [2016-09-25T22:23:21.861253 #443] INFO
-- : Started HEAD "/" \
for 172.18.0.1 at 2016-09-25 22:23:21 +0000
webapp        | App 443 stdout: D, [2016-09-25T22:23:22.110304 #443] DEBUG
-- : ActiveRecord::S\
chemaMigration Load (0.9ms) SELECT "schema_migrations".* FROM "schema_
migrations"
webapp        | App 443 stdout: I, [2016-09-25T22:23:22.157587 #443] INFO
-- : Processing by Rai\
ls::WelcomeController#index as */*
webapp        | App 443 stdout: I, [2016-09-25T22:23:22.157720 #443] INFO
-- : Parameters: {"i\
nternal"=>true}
webapp        | App 443 stdout: I, [2016-09-25T22:23:22.178806 #443] INFO
-- : Rendering /usr/\
local/rvm/gems/ruby-2.3.1/gems/railties-5.0.0.1/lib/rails/templates/rails/
welcome/index.html.erb
webapp        | App 443 stdout: I, [2016-09-25T22:23:22.186551 #443] INFO
-- : Rendered /usr/l\
ocal/rvm/gems/ruby-2.3.1/gems/railties-5.0.0.1/lib/rails/templates/rails/
welcome/index.html.erb (7.5\ms)
webapp        | App 443 stdout: I, [2016-09-25T22:23:22.187048 #443] INFO
-- : Completed 200 OK \
in 29ms (Views: 25.2ms | ActiveRecord: 0.0ms)
webapp        | App 443 stdout:
webapp        | App 443 stdout:
```

Having your logs property configured is important for production tools like Kubernetes and ECS, where inspecting things can be more difficult than on development. You don't want to go inside your production containers and start to follow log files, mainly because you first would have to find the right node that's running that container, and that ruins the whole point of clustered applications.

Pushing the App to DockerHub

In order to be able to run our web application with Kubernetes or ECS, we need to push our image to some registry. Let's use a public repository from DockerHub.

First, make sure you are logged in with your Docker account by running

```
$ docker login
```

That's going to ask you for your DockerHub credentials, and then it's going create a configuration file in a `.docker/config.json` file in your home path. This way, you don't have to log in every time you need to interact with your repositories.

Now go to https://hub.docker.com/ and create a new public repository and name it `webapp`.

The first change we want to make is to create a `.dockerignore` file. I said before that we want to keep our images as light as possible, so, as with GitHub repositories, we can omit some files. Let's create a `.dockerignore` file and add the files we don't want to copy into our images.

```
$ touch .dockerignore
```

And add.

```
# Ignore bundler config.
/.bundle

# Ignore all logfiles and tempfiles.
/log/*
/tmp/*
!/log/.keep
!/tmp/.keep

# Ignore Byebug command history file.
.byebug_history
```

We are basically copying what's inside the `.gitignore` file for a standard Rails app.

Now, let's build an initial version of our application. A practice that I've found to be good and clear is to tag the images with the latest Git commit hash. Of course, we want to use the short version of the commit.

First, we have to tell Git that this is a repository, so in the root of your application run the following:

```
$ git init
$ git add -A
$ git commit -m 'Add dockerized Rails app'
```

Now we can save the short version of the latest commit hash with

```
$ LC=$(git rev-parse --short HEAD)
```

It's very convenient for tagging our image to use that variable instead of copying the string. Let's build our first image using that tag. Replace my username with your DockerHub username and also the repository name in case you didn't use `webapp`.

```
$ docker build -t pacuna/webapp:${LC} .
```

And now push it to the remote repository.

```
$ docker push pacuna/webapp:${LC}
```

Output (truncated):

```
The push refers to a repository [docker.io/pacuna/webapp] 70e47e879cb0:
Pushed
2f00f00b770f: Pushed
6a061828948b: Pushed
87db749cfa82: Pushed
f7feb319b319: Pushed
0945e5099009: Pushing [==> ] 4.929 MB/105.2 MB
90f2b6e5ebff: Pushing [=====================================>] 5.632 kB
20138267dbc3: Pushed
094f4202572b: Pushing [=====================================>] 3.584 kB
f9cdde53d648: Pushing [=====================================>] 3.584 kB
fa4354a3646d: Waiting 662bc11192df: Waiting
...
```

And that's it! Now our image is ready to be deployed with the orchestration framework we want to use.

Those steps for generating an image and pushing it to DockerHub can be automated easily with a bash script. Let's generate a new script file for this task.

```
$ touch push.sh
$ chmod +x push
```

And add the following:

```
#!/bin/sh

LC=$(git rev-parse --short HEAD)
docker build -t pacuna/webapp:${LC} .
docker push pacuna/webapp:${LC}
```

Now, every time we need to push to DockerHub from our local machine we can just run `./push.sh`

Summary

In this chapter, we've seen how to dockerize a Ruby on Rails application using Docker and Docker Compose. We were able to create a new Rails app without having any dependencies but Docker installed on our machine. We also discovered how to keep a smooth development workflow when working with containers. We saw a good way to structure our Docker images by using commit hashes to tag them before pushing them to DockerHub. Finally, we started our first little automation script to build our image and push it to DockerHub using our tagging schema.

CHAPTER 2

Setting Up Tools for Production

One reason a lot of people enjoy using tools like Kubernetes and ECS is their integration with cloud providers. It can be very challenging to configure a cluster by hand, even if you use modern tools like Kubernetes. You have to configure the way your nodes are going to join the cluster and communicate with each other, the service discovery mechanism, and the integration with the extra tools the Cloud offers, like load balancers and persistent storages.

With Kubernetes and ECS, you can launch a cluster without worrying about these issues. Even if you want to build a high-availability cluster, distributed among several regions, you'll have high-level tools to accomplish that without having to dig too heavily into networking issues.

In this section we're going to install an essential dependency that will allow Kubernetes and AWS (Amazon Web Services) to manage our resources for us. This tool is called AWS Command Line Interface (AWS CLI) and allows interaction with all the services available in our AWS account.

If you don't have an AWS account, you can go to the official page (`https://aws.amazon.com/`) and create one now. It's going to be necessary for the following steps. Keep in mind that though you have AWS Free Tier for some resources (with certain limits), we are going to be using resources that don't fall into that category, so you will be billed for the time you run those resources. But don't worry, you can launch as many servers, load balancers, or S3 buckets you want, and you'll have a small bill as long as you delete them all after you're done.

Installing the AWS CLI

The official documentation (`http://docs.aws.amazon.com/cli/latest/userguide/installing.html`) discusses several methods to install the tool according to your operating system (OS). One dependency that you would need on your system is python.

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1_2

I am using Mac Os. If you're also using Mac Os or Linux, you can use the `AWS CLI Bundled Installer` and install AWS CLI on your system with the following commands:

```
$ curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-
bundle.zip"
$ unzip awscli-bundle.zip
$ sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
```

If you're on Windows, you would have to use `pip` and install the `awscli` package. After executing those commands, run

```
$ aws --version
```

Output:

```
aws-cli/1.10.66 Python/2.7.10 Darwin/16.0.0 botocore/1.4.56
```

So the tool is correctly installed.

Configuring the AWS CLI

Before configuring the AWS CLI, you need to get an access key and secret access key for your account. These are personal tokens that will allow third-party applications or the AWS CLI to manage resources on your behalf. In order to get these tokens, go to http://docs.aws.amazon.com/general/latest/gr/managing-aws-access-keys.html and follow the steps. This shouldn't take too long since the entire process is done via the AWS graphical interface.

Once you have your tokens, go back to your console and run

```
$ aws configure

AWS Access Key ID [None]: XXXXXXXXXXXXXXXXXXX
AWS Secret Access Key [None]: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Default region name [None]: us-east-1
Default output format [None]: json
```

And add your own configuration. It's also a good idea to set the default region immediately so you don't have to select it for every command you run.

After running this command, your credentials will be saved under `~/.aws/credentials`:

```
[default]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

And your configuration under `/.aws/config`:

```
[default]
output = json
region = us-east-1
```

In case you wonder, `default` is your default profile. You can have several profiles which are associated with different AWS accounts. So, for example, you can configure a profile for work and another for personal use by using different sections in the files.

```
[default]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

[work]
aws_access_key_id=XXXXXXXXXXXXXXXXXXX
aws_secret_access_key = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The default profile will be used if don't specify any profile when running commands. If you want to run commands using a specific profile, you just have to add the flag `--profile another-profile` to every command.

Now that we have this tool configured, Kubernetes can use it to launch the cluster, and we can also use it for managing our ECS resources.

All the operations you do with the AWS CLI can also be done using the graphical interface. The big advantage of using the CLI is that you can save your commands for avoiding repetition and you can also automate almost everything you want. For example, you can create a load balancer by following a graphical wizard and repeating that process for all the load balancers you need, or you can figure out how to do it by using the AWS CLI, save that command, and run it every time you need a new load balancer.

Tips for Using the AWS CLI

Sometimes you'll have a substantial amount of output after running commands. I can show you a couple of tricks for querying specific fields or filtering by the specific elements we need.

One of the options we'll be using heavily during this book is the `--query` option. With this option you can navigate through the response to a specific part that you want to see. For example, let's say you want to see all your VPC (Virtual Private Cloud) IDs and tags. You can use the `describe-vpcs` command, but if you run it without a query, you'll have too much output. If you want to be more specific and you know which fields you want, you can do something like the following:

```
$ aws ec2 describe-vpcs --region us-east-1 --query="Vpcs[*].
{ID:VpcId,tags:Tags[0]}"
```

That will only return the ID and the tags for your VPCs. The `--query="Vpcs[*].{ID:VpcId,tags:Tags[0]}"` can be interpreted as follows: for all my VPCs (`Vpcs[*]`) fetch the VpcId with an alias of ID (`ID:VpcId`) and the first tag with an alias of tags (`tags:Tags[0]`). That will give a result like the following:

```
[
    {
        "ID": "vpc-e61cec82",
        "tags": null
    },
    {
        "ID": "vpc-a0e9c0c7",
        "tags": {
            "Value": "amazon-ecs-cli-setup-ecs-cluster",
            "Key": "aws:cloudformation:stack-name"
        }
    }
]
```

Much better. Then, once you refine your queries, you can save those commands for posterior use.

Let's see another example. Suppose you have a Database instance. You know its identifier and you want to know its status. You can use the `describe-db-instances` command, but it'll also give a big output. Let's add the following query option:

```
$ aws rds describe-db-instances --db-instance-identifier webapp-postgres
--query 'DBInstances[*].{Status:DBInstanceStatus}'
```

That will only return the database (DB) status

```
[
    {
        "Status": "Running"
    }
]
```

Another important option is the filter option. Normally you'll have a certain set of filters for a specific command. You want to use filters when you have several elements and you need information about a specific one. For example, you need the IDs of a group of subnets for a specific VPC. You want to use the `describe-subnets` command. Without filters it's going to return all your subnets from all VPCs. Let's add a filter to get the ones for a specific VPC.

```
$ aws ec2 describe-subnets --filters "Name=vpc-id,Values=vpc-a0e9c0c7"
--query="Subnets[*].SubnetId"
```

Output:

```
[
    "subnet-3a09f717",
    "subnet-e0906cbb"
]
```

That's going to return the SubnetIds for that specific VPC. For the filter options, you have to pass the name of the filter, which is pre-defined in the documentation, and then the values for applying that filter.

Knowing how to manipulate the output of the AWS CLI can help you a lot. This tool has great documentation. You can use the online documentation or use the `help` command on your console. If at any point you have doubts about a command that we use throughout this book, just jump to its documentation to get more information.

Summary

In this chapter, we prepared the essentials tools we'll need to run our orchestration frameworks. Since this book is focused on Amazon Web Services, we need to have certain tools available on our system and later on our continuous integration (CI) server. Now you should know how to install the AWS CLI and also the basics on how to manipulate the output for the AWS CLI. This is important since we'll be running several different commands to create our AWS resources, and controlling the CLI output can serve as a quick feedback mechanism to collect and inspect data we may need.

CHAPTER 3

Kubernetes

Introduction

Kubernetes defines itself as an open source system for automating deployment, scaling, and management of containerized applications. Google originally designed this product, and it's based on years of experience running containerized architectures. Since this is an open source project, it also has a very big community behind it—one that constantly discusses improvements and new ideas.

Kubernetes Architecture

Like most of the cluster management tools, Kubernetes works with a node-master architecture. All the nodes run an agent that's called **kubelet** and the master runs other components such as APIs (application programming interfaces) and a scheduler.

All the nodes have Docker and are managed from the master systems. **Kubelet** is in charge of managing the **Pods** and the containers in all aspects. We'll talk more about pods later, but now just think of them as an abstraction that groups containers. The nodes also run a network proxy and load balancer. These allow us to have static end points for our container, which are defined through services. This is the part that allows communication inside the cluster between the different services but also from outside the cluster.

On the master, the essential components are `etcd`, which is a key-value data store that keeps the state and allows the coordination between components, and the scheduler, which, as you can imagine, manages the scheduling of the pods.

Another important element is the API server. This component serves the Kubernetes API which provides several REST operations to interact with the cluster. This API defines a lot of different objects you can use to deploy your applications. We aren't going to delve deeply into all of these objects. In this chapter I'll provide a brief overview of the most important ones. Also, Kubernetes is still in development, so new features and new objects appear with every big release.

The objects we are going to look at are Pods, Jobs, Volumes, Replica Sets, Services, and Deployments. These are the basic elements to build almost any application. You can use the other objects in case you need more specific features, like keeping secrets in your cluster, having certain network policies, or having different namespaces.

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1_3

Main Objects

Pods

The pod is the first level of abstraction in Kubernetes. It's just one level above the container. The difference is that you can have several containers running in the same pod. In general, I do not recommend that you do this unless it's a very specific case in which it actually makes sense. Generally, you want to have one container per pod. For our application, we'll use different pods for our web app and for PostgreSQL. The exterior requests can be routed to a pod via services. The pod's metadata will allow us to do a match so the respective service can know where to route the requests.

Following is an example template for a pod for our application:

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    name: webapp
spec:
  containers:
  - image: pacuna/webapp:d15587c
    name: webapp
    env:
    - name: PASSENGER_APP_ENV
        value: production
      ports:
      - containerPort: 80
        name: webapp
      imagePullPolicy: Always
```

As you can see, we always start by declaring the API version we want to use. Then we declare the type of object. The good thing about declaring the type in the template is that then you can run any template with the same command, and Kubernetes will figure out what type of object you're creating. The labels will help the replica set to identify which pods should be associated with it. All the Kubernetes architecture is based on metadata. A pod will have some metadata that will allow it to be related to a replica set. Then the replica set will have metadata that will allow a service to route requests to it. Finally, a deployment will also know which replica sets are associated with it via metadata.

The specifications section is pretty similar to a Docker Compose template. We declare the image we need, the environmental variables, the ports we want to expose, and extra information. In this case we're adding an `imagePullPolicy` of always, so the image is always pulled.

Replica Set

This object manages our desired number of replicas for a pod. For example, we can specify that we need at least three replicas for a certain pod. Then the replica set will make sure that these three replicas are always up. If one pod dies for any reason, the replica set will create a new one. In previous Kubernetes versions, the Replication Controller was the object that provided these functionalities. Even though this is still supported, Kubernetes defines the replica set as the next-generation replication controller. And the only difference is that the replica set supports different types of selectors to filter certain objects.

Following is an example replica set for our application:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: webapp
        tier: frontend
    spec:
      containers:
      - image: pacuna/webapp:d15587c
        name: webapp
        env:
        - name: PASSENGER_APP_ENV
          value: production
        ports:
        - containerPort: 80
          name: webapp
        imagePullPolicy: Always
```

The replica set also uses metadata in order to identify the pods in the set. As you can see, you can add the pod declaration inside the replica set specification. This type of embed is typical in Kubernetes configurations and can help you to avoid having too many files that you need to update when a new image is available for your app. Later you'll see that with just one file we can launch an entire system. An important part of the specification of a replica set is the `replicas` property. As you can imagine, this property controls the number of pods for this set.

Jobs

A job is simply a pod with a well-defined task to accomplish. The action that a job does better than a pod has to do with task completion. Every time you launch a job, if for some reason the task fails, the job will stop and it'll start again until the task finishes. If you use a regular pod to run your task, you won't be sure your task gets completed, because if the pod dies, it won't start again. Jobs are supereffective for fetching data from external sources, completing long data processing tasks, completing long computing tasks, and so on. But they are also useful to separate our application maintenance tasks and our actual web application containers. For example, in a typical Rails application we need to run migration and precompile assets before a new deploy. Instead of running these tasks in the same web app container, we can run them by launching independent jobs and that way avoid collisions in case we launch several web application pods.

Following is an example for a job that runs migrations for our web application:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: setup
spec:
  template:
    metadata:
      name: setup
    spec:
      containers:
      - name: setup
          image: pacuna/webapp:ec4421
          command: ["./bin/rails", "db:migrate", "RAILS_ENV=production"]
          env:
          - name: PASSENGER_APP_ENV
            value: production
        restartPolicy: Never
```

The specification is pretty similar to a pod specification. We have to use the same image that we have for our application, since it need to be up to date with the latest migrations and also needs the configuration for writing to the database. As I mentioned previously, this job will run until it completes the task. After completion it will never be restarted until you run the task again. We make sure of that by adding a `restartPolicy` of Never.

Volumes

This object is pretty similar to what we know as volumes in Docker. It allows us to add a persistent object to our pods. Remember that when a container crashes, it gets restarted, by **kubelet** in this case, but all its disk files will be lost. And that's because Docker filesystems are temporary by default. Volumes also solve the issue of sharing data between different containers. These containers can mount the same volume to access the files inside the volume.

Kubernetes's big advantage is that it has a tight integration with cloud providers like AWS and GCE. This integration makes possible the use of storage units like EBS (Elastic Block Store) in the case of AWS. That's very important, since in some cases, simple volume mounts on the same host are not enough.

For example, let's say you have a database running in a container, and you want to mount the data folder so your data is persisted on the host. What happens if you run a new container version of your database and it gets deployed on a new node? Then the volume mount won't work, because the data is persisted on another machine. On the other hand, if you mount a volume using something like EBS, then it's always going to be that same external storage unit and you won't have any issues if your container launches on a different machine every time.

For example, following is the specification for a PostgreSQL container with an associated EBS volume that was created previously:

```
spec:
  containers:
  - image: postgres:9.5.3
    name: postgres
    env:
    - name: POSTGRES_PASSWORD
      value: mysecretpassword
    - name: POSTGRES_USER
      value: webapp
    - name: POSTGRES_DB
      value: webapp_production
    - name: PGDATA
      value: /var/lib/postgresql/data/pgdata
    ports:
    - containerPort: 5432
      name: postgres
    volumeMounts:
      - name: postgres-persistent-storage
        mountPath: /var/lib/postgresql/data
  volumes:
    - name: postgres-persistent-storage
      awsElasticBlockStore:
        volumeID: vol-fe268f4a
        fsType: ext4
```

The relevant part is

```
volumeMounts:
  - name: postgres-persistent-storage
    mountPath: /var/lib/postgresql/data
```

```
volumes:
- name: postgres-persistent-storage
  awsElasticBlockStore:
    volumeID: vol-fe268f4a
    fsType: ext4
```

The `volumes` part declares the volume we want to use—in this case we use the ID that AWS assigned to that EBS. Then, inside the pod's specification, we can add a `volumeMounts` section indicating the name of the volume we want to use and the path that we want to be persisted.

Services

A service is going to give us a static end point for our replica sets and pods. Every time you launch a new version of your pods, their IP (Internet Protocol) address will change. It doesn't matter if it is internal or external, we need a mechanism to route requests to those pods. A service can be a typical load balancer, or we can just declare that we want to expose a port in our cluster that routes requests to the same group of pods. Every container orchestration framework has a mechanism to handle this issue. It's necessary if you want to have a public end point for your application.

Kubernetes also has a nice integration with AWS ELBs. You can declare that your service should be a load balancer and Kubernetes will create and associate one automatically.

Following is an example for a load balancer service for our web app:

```
apiVersion: v1 kind:
Service
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  ports:
    - port: 80
 selector:
   app: webapp
   tier: frontend
type: LoadBalancer
```

Pretty simple, besides the metadata, we indicate we want to expose the port 80 and the type should be a load balancer. As I said before, Kubernetes will create a new ELB (Elastic Load Balancer) in our AWS account and it'll be associated with the pods that match the metadata with this service.

Deployments

Kubernetes introduced deployments in one of its latest releases. It can be a little bit confusing at first, but when you start to use them, you will see their value and how they can make your update process run very smoothly. In a deployment declaration, you can have your replica set and pod specifications declared together. The deployment will be in charge of running updates to the state of these elements. So you can update the image in your deployment, and the respective replication set and pods will be updated as well.

Following is an example deployment for PostgreSQL:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: webapp
spec:
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: webapp
        tier: postgres
    spec:
      containers:
      - image: postgres:9.5.3
        name: postgres
        env:
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
        - name: POSTGRES_USER
          value: webapp
        - name: POSTGRES_DB
          value: webapp_production
        ports:
        - containerPort: 5432
          name: postgres
```

This deployment template contains a replica set and pod in its specification. With this individual file, you can have a pod and a replica set and you can run updates to both with only one command. We have to add a couple of extra properties, like the strategy to run updates, which in this case should be **Recreate** since we don't want to have more than one of this pod running at the same time. However, you can also use `RollingUpdate` but it will do the opposite.

Using deployments is recommended over using replication sets and pods independently.

Structuring the Files

In order to keep a better application structure, let's add some folders for the Kubernetes configuration.

We'll use a kube folder, and subfolders for every Kubernetes type we need. For now, let's create deployments and jobs subfolders.

```
$ mkdir -p kube
$ mkdir -p kube/deployments
$ mkdir -p kube/jobs
```

Templates

PostgreSQL

Let's create a deployment file for the PostgreSQL service.

```
$ touch kube/deployments/postgres-deployment.yaml
```

And add the following:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    app: webapp
spec:
  ports:
    - port: 5432
  selector:
    app: webapp
    tier: postgres
  clusterIP: None
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: webapp
spec:
  template:
    metadata:
      labels:
        app: webapp
        tier: postgres
```

```
    spec:
      containers:
      - image: postgres:9.5.3
        name: postgres
        env:
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
        - name: POSTGRES_USER
          value: webapp
        - name: POSTGRES_DB
          value: webapp_development
        ports:
        - containerPort: 5432
          name: postgres
```

That's kind of a long YAML (YAML Ain't Markup Language) file (and it'll get longer), but keep in mind that the file contains all the necessary elements that we need for the PostgreSQL deployment. It'll generate the pod, the replica set that's going to manage the pod, and also the service so our web application and the setup container can reach it by using the same alias we used in development with Docker Compose.

Let's go section by section.

```
apiVersion: v1
kind: Service
metadata:
name: postgres
labels:
  app: webapp
spec:
ports:
  - port: 5432
selector:
  app: webapp
  tier: postgres
clusterIP: None
```

The first section of the file is the service declaration. In the deployment file we can declare more than one kind, and we separate them by using `---`.

This service has a name, `postgres`, which means that we are going to be able to use that alias for communicating with it from our cluster. Kubernetes uses its own DNS (Domain Name System) mechanism to allow communication between services running on different nodes by using their aliases. We are also defining a label that can be useful for filtering.

The spec section will tell the service where it should be routing the requests. In this case it's going to match two selectors, `app` and `tier`, with the value `webapp`, which is the main context, and the specific tier, which is `postgres`. The `tier` is the selector that's going to differentiate the web application containers and the PostgreSQL container, since both are going to have an `app` selector with the value `webapp` but a different `tier` selector value.

The purpose of the app selector is to able to have several different applications running in the cluster; for example, if you have another application named users, you can have the same postgres tier but a different app label.

We are also indicating that the communication port should be 5432, which is the standard for PostgreSQL. And finally, we are telling Kubernetes that this service should be accessed only from inside the cluster by using clusterIP set to None. That means you won't be able to hit this end point from outside the cluster.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: webapp
spec:
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: webapp
        tier: postgres
    spec:
      containers:
      - image: postgres:9.5.3
        name: postgres
        env:
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
        - name: POSTGRES_USER
          value: webapp
        - name: POSTGRES_DB
          value: webapp_development
        ports:
        - containerPort: 5432
          name: postgres
```

The second section corresponds to the deployment itself.

We start by adding the same metadata we used for the service and then the specifications. Then, for the template of the specification, we have to specify the match we need for this replica set and pods. As I mentioned earlier, the service will be looking for two labels—app and tier–and they must match this metadata. We also indicate that we want to use a Recreate strategy type. This means that we want to kill all the existing pods for this deployment before launching new ones. If you don't specify a strategy, the deployment will use a default, which is RollingUpdate. RollingUpdate won't kill the old Pods before launching the new ones, so you can better control your update process. That works well with web applications, but you can see why it would be a problem when using containers for something like databases. You don't want to have two copies of the same database container running at the same time.

Finally, we define the specification for the containers. This is pretty similar to the Docker Compose structure and it contains the image, the environmental variables, the port, and the container name. Notice that we're still using the variables for development. Later we'll switch this to production once we configure our application to run on that environment.

That's it for the PostgreSQL deployment for now. Later we'll add another section for the persistence storage. We are skipping persistence storage for now because we first want to test this template with a local tool, and the way you manage volumes with Kubernetes doesn't allow us to test persistence storage locally.

Setup Container

In the case of the setup container, we don't need a replica set, service, or deployment. We only need a pod that gets the job done. For that we are going to use a Kubernetes job (http://kubernetes.io/docs/user-guide/jobs/). Think of a job as a special kind of pod that was meant to be used for these kinds of tasks. It's going to create a pod that'll run the command we indicate and it'll end after the container finishes.

The syntax for writing a job template is quite simple. Let's create the file

```
$ touch kube/jobs/setup-job.yaml
```

And add the following configuration, replacing the image with the one you pushed to DockerHub. Remember we are using the latest commit hash for tagging:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: setup
spec:
  template:
    metadata:
      name: setup
    spec:
      containers:
      - name: setup
        image: pacuna/webapp:57514be
        command: ["/bin/bash", "./setup.sh"]
        env:
        - name: PASSENGER_APP_ENV
          value: development
      restartPolicy: Never
```

This pod has a very simple definition. It has a `restartPolicy` of never, so once it finishes **kubelet** doesn't restart it. All failed containers are restarted by `kubelet` using an exponential back-off delay. If a `restartPolicy` is not defined, the default is `Always`.

The `containers` section contains the specs for the image that we just pushed to DockerHub. Similar to what we did with Docker Compose, we set the necessary environmental variables and a command instruction to run our custom entry point. As you can see, in Kubernetes there isn't any `entrypoint` instruction but a `command` key that behaves in a similar fashion. Remember that our entry point for this container should be our `setup.sh` file that runs the migrations once the PostgreSQL container is ready.

Web Application

This deployment file is also going to have two parts: a service and a deployment.

```
$ touch kube/deployments/webapp-deployment.yaml
```

And add the following template:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  ports:
    - port: 80
  selector:
    app: webapp
    tier: frontend
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: webapp
        tier: frontend
    spec:
      containers:
      - image: pacuna/webapp:57514be
        name: webapp
```

```
        env:
        - name: PASSENGER_APP_ENV
          value: development
        ports:
        - containerPort: 80
          name: webapp
        imagePullPolicy: Always
```

Again, let's go by part. First, the service definition is pretty similar to the one we have for the PostgreSQL deployment. The difference here is that we are using a new set of selectors. The PostgreSQL uses a tier named `postgres`, and this one uses a tier named `frontend`. This way the service can route the requests to the correct replica set, which has those same labels.

Another difference is that we are using a `type` for the service. Locally we can use the type `NodePort`. Which is a way of exposing a port for accessing the application from outside the cluster. In production we'll use a `LoadBalancer` type, a type that we don't have for our local cluster. The difference between these two types is that the `NodePort` will expose a random port from the cluster for accessing the application. On the other side, the `LoadBalancer` type will use the provider's API (AWS or GCE) for creating a real load balancer in your account. And this load balancer will have a static DNS we can use for connecting to the service, and also will balance the requests among all the pod replicas we declare in the specs.

Then we have the deployment section. This section contains the metadata with the respective name and label and the specification section. We are indicating that we want to have three pods for this deployment. That means there are going to be three containers running our application, and this service will be routing the traffic to those three pods. We then declare the labels so the service can localize the respective replica set.

Finally we have the container specifications. We are using the image we pushed to DockerHub (replace it with yours), passing the `PASSENGER_APP_ENV` variable for setting the Rails environment, exposing the port, and adding an `imagePullPolicy` in order to always force the pull.

Your final structure for the deployment files should be something like the following:

```
kube
├── deployments
│   ├── postgres-deployment.yaml
│   └── webapp-deployment.yaml
└── jobs
    └── setup-job.yaml
```

Now it's time to try those templates on a local cluster. For that we are going to use a very handy tool called **Minikube**.

Minikube

According to the official Minikube repository (see https://github.com/kubernetes/minikube):

> *Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.*

Before Minikube, running Kubernetes locally was difficult. You had some alternatives like running a cluster using Vagrant or just use plain Docker. I believe that Minikube gives you a much better experience. Most of the setup is pretty transparent for the user and like almost all of the Kubernetes tooling, it just works.

We'll use Minikube in order to test our Kubernetes configuration files without having to launch test servers in the cloud. That's a very big step in the container orchestration world. Even though Docker reduces the friction between development and production environments, you still need to make the transition between your local **deployment** and your real production deployment.

Now, don't confuse testing your cluster locally with local development. Minikube isn't yet a development tool but a tool for testing your cluster's configuration before shipping it to production. You still need to use Docker and Docker Compose or any other tool you use daily to work on your code.

You can find the installation instructions for Minikube in the official repository (https://github.com/kubernetes/minikube). Check the releases page to find the instructions related to your operating system (OS). In general, the installation should be pretty straightforward. The easiest way to install Minikube is just to download the Go binary and then add it to your path. An important requirement for Minikube is a virtualization mechanism. You have different options such as `xhyve`, `VirtualBox`, or `KVM` depending on your OS.

The version I have on my machine is v.0.10.0 and I'll be using the `xhyve` driver. You can use whatever is best for your platform.

You also need the **kubectl CLI** (Command Line Interface) tool for launching the services and deployments. See http://kubernetes.io/docs/getting-started-guides/minikube/#download-kubectl for the instructions to install kubectl. This CLI tool communicates with our cluster in order to run containers and services. It has a pretty solid integration with cloud services like Amazon AWS and Google's GCE. This tool will be our main channel of interaction with our cluster.

Right now I'm using version 1.4.0 for kubectl. Make sure you're using that version or a newer one.

Once you have Minikube and kubectl ready to go, you can create your local cluster by running (add the **–vm-driver** flag if you want to use a specific VM provider and **–kubernetes**-version for a specific version of Kubernetes).

In my case I'm using Mac OS with the xhyve driver and since I have the version 1.4.0 for my client (kubectl), I'll use the latest version available with Minikube for my cluster.

```
$ minikube start --vm-driver=xhyve --kubernetes-version="v1.4.0-beta.10"
```

That can take a few minutes the first time. Once it finishes, you should see an output similar to the following:

```
Starting local Kubernetes cluster...
Kubectl is now configured to use the cluster.
```

That indicates that your local Kubernetes cluster is running. Kubectl will now be configured to interact with your cluster. This is because kubectl can have different contexts, which means you can have different clusters configured at the same time and switch the context for interacting with each one. In this case when we run `minikube start` it'll also set the current context to minikube.

Minikube gives you a handful set of commands for interacting with the cluster. For example, you can run

```
$ minikube ip
```

Output:

```
192.168.64.7
```

This is the IP address for your cluster. You can also run

```
$ minikube dashboard
```

Which will open your default browser and take you to the Kubernetes UI (user interface) where you can see all the workloads running in the cluster. Right now we shouldn't have any. You can also run tasks and services from this Dashboard. I don't think many people would actually use the UI as a main way of managing their services, but it's still a nice to have this element for monitoring and have a quick view of what's happening in our cluster.

Running Our Templates with Minikube

Sadly, Kubernetes currently doesn't have a dependency mechanism like Docker Compose does. So we have to create our resources in the correct order.

The first **kubectl** command you should know is `create -f`. This command can save us a lot of headaches from trying to memorize different syntaxes for running different Kubernetes kinds. You can pass any template as an argument to this command, and as long as you have the `Kind` defined in the template, kubectl will know what to do with it.

Let's start with the PostgreSQL deployment template.

Run the following command from the root of the application:

```
$ kubectl create -f kube/deployments/postgres-deployment.yaml
```

Output:

```
service "postgres" created
deployment "postgres" created
```

We can describe our new deployment with

```
$ kubectl describe deployment postgres
```

Output:

```
Name:                   postgres
Namespace:              default
CreationTimestamp:      Mon, 26 Sep 2016 20:21:45 -0300
Labels:                 app=webapp
Selector:               app=webapp,tier=postgres
Replicas:               1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           Recreate
MinReadySeconds:        0
OldReplicaSets:         <none>
NewReplicaSet:          postgres-1747921911 (1/1 replicas created)
Events:
  FirstSeen  LastSeen  Count  From              SubobjectPath   Type
  ---------  --------  -----  ----              -------------   --------
  4s         4s          1   {deployment-controller}  Normal  921911 to 1
```

If you run `minikube dashboard` again, you should see the new workloads under the correspondent sections.

This deployment generates a service, a replica set, and a pod. If you want to see the events for the pod, you can run

```
$ kubectl describe Pod postgres
```

And you'll have a long output that describes the pod specifications and also the events in case something goes wrong. Following is an output for the events section:

```
Events:
  FirstSeen LastSeen Count From               SubobjectPath   Type
  --------- -------- ----- ----               -------------   --------
  5m        5m       1 {default-scheduler}    Normal          6t to minikube
  5m        5m       1 {kubelet minikube} spec.containers{postgres} Normal
  .5.3" already present on machine
  5m        5m       1 {kubelet minikube} spec.containers{postgres} Normal
  ker id b1c664c98251; Security:[seccomp=unconfined]
  5m        5m       1 {kubelet minikube} spec.containers{postgres} Normal
  ker id b1c664c98251
```

As you can see, all the events are pretty standard. The image is pulled from DockerHub and the container is launched using the specification we gave it.

Now that we have our database running, let's launch the setup job that's going to run the migrations for the application.

```
$ kubectl create -f kube/jobs/setup-job.yaml
```

Output:

```
job "setup" created
```

This can take a few minutes the first time, since it has to pull the entire image from DockerHub.

To see what's happening with our job, we can filter the job's pods by the job's name and then run the logs command for that pod.

First let's get the name of the pod. The name we used for the job was `setup`.

```
$ Pods=$(kubectl get Pods --selector=job-name=setup --output=jsonpath=
{.items..metadata.name})
```

Now if you run `echo $Pods` you'll get the name of the associated pod.

Let's see the logs (if you see an error with ContainerCreating status, wait for a few more minutes).

```
$ kubectl logs $Pods
```

Output:

```
Waiting PostgreSQL to start on 5432...
PostgreSQL started
D, [2016-09-26T23:29:37.704679 #6] DEBUG -- :  (103.0ms) CREATE TABLE
"schema_migrations" ("versi\
on" character varying PRIMARY KEY)
D, [2016-09-26T23:29:37.776729 #6] DEBUG -- :  (67.1ms) CREATE TABLE "ar_
internal_metadata" ("key\
" character varying PRIMARY KEY, "value" character varying, "created_at"
timestamp NOT NULL, "update\
d_at" timestamp NOT NULL)
D, [2016-09-26T23:29:37.778284 #6] DEBUG -- :  (0.3ms) SELECT pg_try_
advisory_lock(19324101055242\
39390);
D, [2016-09-26T23:29:37.822212 #6] DEBUG -- :  ActiveRecord::SchemaMigration
Load (0.6ms) SELECT "\
schema_migrations".* FROM "schema_migrations"
I, [2016-09-26T23:29:37.839451 #6] INFO -- : Migrating to CreateArticles
(20160925220117)
```

```
D, [2016-09-26T23:29:37.841086 #6] DEBUG -- : (0.2ms) BEGIN
== 20160925220117 CreateArticles: migrating ================================
-- create_table(:articles)
D, [2016-09-26T23:29:37.908021 #6] DEBUG -- : (63.7ms) CREATE TABLE
"articles" ("id" serial prim\
ary key, "title" character varying, "body" text, "created_at" timestamp NOT
NULL, "updated_at" times\
tamp NOT NULL)
   -> 0.0652s
== 20160925220117 CreateArticles: migrated (0.0656s) =======================

D, [2016-09-26T23:29:37.914849 #6] DEBUG -- : SQL (0.5ms) INSERT INTO
"schema_migrations" ("versi\
on") VALUES ($1) RETURNING "version" [["version", "20160925220117"]]
D, [2016-09-26T23:29:37.937591 #6] DEBUG -- : (22.2ms) COMMIT
D, [2016-09-26T23:29:37.943480 #6] DEBUG -- : ActiveRecord::InternalMetadata
Load (0.3ms) SELECT \
 "ar_internal_metadata".* FROM "ar_internal_metadata" WHERE "ar_internal_
metadata"."key" = $1 LIMIT \
$2 [["key", :environment], ["LIMIT", 1]]
D, [2016-09-26T23:29:37.947971 #6] DEBUG -- : (0.1ms) BEGIN
D, [2016-09-26T23:29:37.951697 #6] DEBUG -- : SQL (2.5ms) INSERT INTO "ar_
internal_metadata" ("ke\
y", "value", "created_at", "updated_at") VALUES ($1, $2, $3, $4) RETURNING
"key" [["key", "environm\
ent"], ["value", "development"], ["created_at", 2016-09-26 23:29:37 UTC],
["updated_at", 2016-09-26 \
23:29:37 UTC]]
D, [2016-09-26T23:29:37.963675 #6] DEBUG -- : (11.4ms) COMMIT
D, [2016-09-26T23:29:37.964502 #6] DEBUG -- : (0.4ms) SELECT pg_advisory_
unlock(1932410105524239\390)
D, [2016-09-26T23:29:37.970691 #6] DEBUG -- : ActiveRecord::SchemaMigration
Load (0.2ms) SELECT "\
schema_migrations".* FROM "schema_migrations"
D, [2016-09-26T23:29:37.988811 #6] DEBUG -- : (1.8ms) SELECT
t2.oid::regclass::text AS to_table,\
a1.attname AS column, a2.attname AS primary_key, c.conname AS name,
c.confupdtype AS on_update, c.c\
onfdeltype AS on_delete
FROM pg_constraint c
JOIN pg_class t1 ON c.conrelid = t1.oid
JOIN pg_class t2 ON c.confrelid = t2.oid
JOIN pg_attribute a1 ON a1.attnum = c.conkey[1] AND a1.attrelid = t1.oid
JOIN pg_attribute a2 ON a2.attnum = c.confkey[1] AND a2.attrelid = t2.oid
JOIN pg_namespace t3 ON c.connamespace = t3.oid
```

```
WHERE c.contype = 'f'
  AND t1.relname = 'articles'
  AND t3.nspname = ANY (current_schemas(false))
ORDER BY c.conname
```

Cool! So the container ran the migrations and then died. Just what we wanted. If you want to make sure the container died, you can list the pods with

```
$ kubectl get Pods
```

Output:

```
NAME                        READY   STATUS    RESTARTS   AGE
postgres-1747921911-h516t   1/1     Running   0          10m
  info: 1 completed object(s) was(were) not shown in Pods list. Pass --show-
all to see all objects.
```

Just the PostgreSQL Pod and one completed pod, which was the migrate job. Now, let's run the last piece, the web application.

```
$ kubectl create -f kube/deployments/webapp-deployment.yaml
```

Output:

```
service "webapp" created
deployment "webapp" created
```

Don't worry if you get a warning about port security. That message would be relevant only if we were using this service type on a production cluster.

This command should run much faster than the previous one, since we already have the image on the cluster. Let's check that by running

```
$ kubectl get Pods
```

Output:

```
NAME                        READY     STATUS     RESTARTS    AGE
postgres-1747921911-h516t   1/1       Running    0           12m
webapp-78833492-68glu       1/1       Running    0           53s
webapp-78833492-op9v6       1/1       Running    0           53s
webapp-78833492-riyfd       1/1       Running    0           53s
  info: 1 completed object(s) was(were) not shown in Pods list. Pass --show-
all to see all objects
```

There you can see the three replicas we declared for the web application running along the PostgreSQL pod.

Let's check the logs for one of those pods.

```
$ kubectl logs webapp-78833492-68glu
```

Output:

```
*** Running /etc/my_init.d/00_regen_ssh_host_keys.sh...
*** Running /etc/my_init.d/30_presetup_nginx.sh...
*** Running /etc/rc.local...
*** Booting runit daemon...
*** Runit started as PID 8
ok: run: /etc/service/nginx-log-forwarder: (pid 27) 0s
Sep 26 23:32:57 webapp-78833492-68glu syslog-ng[19]: syslog-ng starting up;
version='3.5.6'
[ 2016-09-26 23:32:58.9714 28/7f31c0c0b780 age/Wat/WatchdogMain.cpp:1291 ]:
Starting Passenger watch\
dog...
[ 2016-09-26 23:32:58.9989 31/7f7166c2d780 age/Cor/CoreMain.cpp:982 ]:
Starting Passenger core...
[ 2016-09-26 23:32:58.9990 31/7f7166c2d780 age/Cor/CoreMain.cpp:235 ]:
Passenger core running in mul\
ti-application mode.
[ 2016-09-26 23:32:59.0071 31/7f7166c2d780 age/Cor/CoreMain.cpp:732 ]:
Passenger core online, PID 31
[ 2016-09-26 23:32:59.0285 38/7f0462922780 age/Ust/UstRouterMain.cpp:529 ]:
Starting Passenger UstRo\
uter...
[ 2016-09-26 23:32:59.0359 38/7f0462922780 age/Ust/UstRouterMain.cpp:342 ]:
Passenger UstRouter onli\
ne, PID 38
```

So it seems the passenger process started with no errors.

If we want to see our application in action, there's a very nice Minikube command that will takes us to our service. Run

```
$ minikube service webapp
```

And that should take you to the cluster's IP address and the port that was assigned to our service. There's where our application lives. Every time you hit that URL (uniform resource locator), the traffic gets load balanced among the replicas you declare for the service, which in this case is the three pods.

Let's find our service address and make a post request to our articles end point. First we need our Minikube VM IP.

```
$ minikube ip
```

Output:

```
192.168.64.7
```

And now the port the cluster assigned to our service.

```
$ kubectl describe service webapp
```

Output:

```
Name:                webapp
Namespace:           default
Labels:              app=webapp
Selector:            app=webapp,tier=frontend
Type:                NodePort
IP:                  10.0.0.122
Port:                <unset>       80/TCP
NodePort:            <unset>       30028/TCP
Endpoints:           172.17.0.4:80,172.17.0.5:80,172.17.0.6:80
Session Affinity:    None
No events.%
```

So, in my case I know that the service should be reachable at `http://192.168.64.7:30028`. In your case, you have to find your `NodePort` in the service description in order to find the exposed port.

Let's make a curl POST request to the end point:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"my first
article","body":"Lorem ipsum dolor sit amet, consectetur adipiscing
elit..."}' http://192.168.64.7:30028/articles
```

Output:

```
{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit..\
.","created_at":"2016-09-26T23:36:19.285Z","updated_at":"2016-09-
26T23:36:19.285Z"}%
```

Great! We have our application fully running in our local Kubernetes cluster.

The transition to a production cluster will not be hard at all. We just need to configure two more elements in our architecture: the load balancers and the volume for persisting the data.

There are also some minor changes we need to make to our application to run on production. For example, we'll need a secret token for production and also to configure our database.

Launching an AWS Kubernetes Cluster

Right now if we run

```
$ kubectl config current-context
```

We get:

```
minikube
```

That's because when we installed Minikube, we created a specific context so we could interact with our local cluster with kubectl.

After we create our new AWS cluster, the Kubernetes installation script will create a new context for us, so we can switch back and forth between Minikube and the production cluster.

Remember that in order to run all these steps, you need to have an AWS account and you have to configure the AWS CLI to be used by Kubernetes. For more information you can go to the Setting up tools for Production section for this book (the URL for a book's web page is www.apress.com/PrintISBN).

The getting started guide for AWS (http://kubernetes.io/docs/getting-started-guides/aws/) indicates that we can run a simple command for launching a cluster with some default settings. You have to keep in mind that these resources are not in the Free Tier, so you may want to delete the entire cluster once you no longer need it. That way you only are going to pay according to the number of hours your instances were up.

The documentation also indicates that:

By default, the script provides a new VPC and a four-node k8s cluster in us-west-2a (Oregon) with EC2 instances running on Debian and then mentions that you can override this configuration by setting the following environmental variables in your system:

```
KUBE_AWS_ZONE

NUM_NODES

MASTER_SIZE

NODE_SIZE

AWS_S3_REGION

AWS_S3_BUCKET

INSTANCE_PREFIX
```

In our case, four nodes can be a little too much just for this simple API and the Database. Let's just use two small instances. In your profile file (`.bashrc` or `.zshrc`) add the following:

```
export NUM_NODES=2
export NODE_SIZE=t2.small
```

And now source that file with (in my case)

```
$ source ~/.zshrc
```

Now we can run the Kubernetes script that will launch a cluster in our account:

```
$ export KUBERNETES_PROVIDER=aws; curl -sS https://get.k8s.io | bash
```

That can take a while. The script will download Kubernetes (which is kind of heavy) and then it'll launch the nodes and all the necessary elements for the cluster. It's going to create Security Groups, a VPC, an S3 Bucket, and the EC2 instances among other stuff.

Once it's finished, at the end of the output you should see something like the following:

```
Kubernetes master is running at https://52.32.34.173
Elasticsearch is running at https://52.32.34.173/api/v1/proxy/namespaces/
kube-system/services/elasti\ csearch-logging
Heapster is running at https://52.32.34.173/api/v1/proxy/namespaces/kube-
system/services/heapster Kibana is running at https://52.32.34.173/api/v1/
proxy/namespaces/kube-system/services/kibana-logging KubeDNS is running at
https://52.32.34.173/api/v1/proxy/namespaces/kube-system/services/kube-
dns kubernetes-dashboard is running at https://52.32.34.173/api/v1/proxy/
namespaces/kube-system/services\
/kubernetes-dashboard
Grafana is running at https://52.32.34.173/api/v1/proxy/namespaces/kube-
system/services/monitoring-g\ rafana
InfluxDB is running at https://52.32.34.173/api/v1/proxy/namespaces/kube-
system/services/monitoring-\ influxdb
```

There you can see all the different services that Kubernetes installed in our cluster and you'll also find your cluster's IP address. It also has added the configuration credentials for this cluster in the `~/.kube/config` file. There you should find a segment similar to the following:

```
  name: aws_kubernetes-basic-auth
  user:
  password: xxxxxxxxxxxxxxxx
  username: admin
```

If you inspect the file, you should also see the Minikube sections. This file is what allows kubectl to interact with different clusters by switching context. If you now run

```
$ kubectl config current-context
aws_kubernetes
```

That's the new context Kubernetes created for us.

Visit your cluster's IP address and use those credentials for login to the cluster.

Once you're logged in, you'll first see a list of end points you can access from there. The dashboard is under the `/ui` namespace. So in my case I can visit https://52.32.34.173/ui and that'll take me to my production cluster's dashboard.

Now we just need to prepare our templates for production.

Running the Templates in Production

First, we need to add a secret token for the production environment. In the root of the application, run

```
$ docker-compose run --rm webapp bin/rake secret RAILS_ENV=production

Starting webapp_postgres_data_1
Starting webapp_webapp_setup_1
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXX\ XXXXXXXXXXXXXXXXXXXXXXXXXXX
```

And copy that token to the `config/secrets.yml` file under the production section.

```
production:
  secret_key_base: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Now open the `config/database.yml` file and add the following values under the production section:

```
production:
  <<: *default
  host: postgres
  database: webapp_production
  username: webapp
  password: mysecretpassword
```

Notice here we are using a new database named `webapp_production`. Right now we are declaring the database credentials in the postgres deployment template. Open the `kube/deployments/postgres.yaml` and change the environmental variables to the following:

```
env:
- name: POSTGRES_PASSWORD
  value: mysecretpassword
- name: POSTGRES_USER
  value: webapp
- name: POSTGRES_DB
  value: webapp_production
```

Let's run this first template.

```
$ kubectl create -f kube/deployments/postgres-deployment.yaml
```

Output:

```
service "postgres" created deployment "postgres" created
```

Let's describe our service.

```
$ kubectl describe service postgres
```

Output:

```
Name:                postgres
Namespace:           default
Labels:              app=webapp
Selector:            app=webapp,tier=postgres
Type:                ClusterIP
IP:                  None
Port:                <unset> 5432/TCP
Endpoints:           10.244.0.7:5432
Session Affinity:    None
No events.%
```

So it seems the service was correctly deployed.

Let's continue with some changes to the setup container job template. First, we must change the `PASSENGER_APP_ENV` environmental variable so the Rails environment is production.

```
env:
- name: PASSENGER_APP_ENV
  value: production
```

Next, the `setup.sh` we currently have is migrating our development database, since it doesn't have the `RAILS_ENV=production` declaration along the command. Let's generate a new setup file for production environment

```
$ touch setup.production.sh
$ chmod +x setup.production.sh
```

And add the following:

```
#!/bin/sh

echo "Waiting PostgreSQL to start on 5432..."

while ! nc -z postgres 5432; do
   sleep 0.1
done

echo "PostgreSQL started"

bin/rails db:migrate RAILS_ENV=production
```

As you can see, the file is practically the same. The only distinction is that we are using the production value for the RAILS_ENV variable. Since we have a new entry point, let's add this new file to the entry point of the setup job command.

```
command: ["/bin/bash", "./setup.production.sh"]
```

Before running the setup template, let's update our Docker image using the script push.sh we created previously:

```
$ git add .
$ git commit -m 'add production templates'
$ ./push.sh
```

Output (truncated):

```
Sending build context to Docker daemon 24.32 MB
Step 1 : FROM phusion/passenger-ruby23:0.9.19
 ---> 6841e494987f
Step 2 : ENV HOME /root
 ---> Using cache
 ---> e70985107acc
Step 3 : CMD /sbin/my_init
 ---> Using cache
 ---> babd8b525225
Step 4 : RUN apt-get update && apt-get install -y -o Dpkg::Options::="--
force-confold" netcat
 ---> Using cache
 ---> 74da8c84b454
Step 5 : RUN rm -f /etc/service/nginx/down
...
The push refers to a repository [docker.io/pacuna/webapp] c3dc4ebe27b6:
Pushed
d29dfd2261fa: Pushing [================>                  ] 7.566 MB/23.13 MB
3f3247bf1fef: Pushed
e5d0c4cf80b0: Pushing [================================> ] 15.17 MB/23.13 MB
f7feb319b319: Layer already exists
0945e5099009: Layer already exists
90f2b6e5ebff: Layer already exists
20138267dbc3: Layer already exists
...
```

Now that our image is up to date, let's modify the setup job using the latest tag we just pushed. You can get the latest tag running `git rev-parse --short HEAD` or running `docker images` and see the latest generated image for your application.

```
$ git rev-parse --short HEAD
915685c

$ docker images

REPOSITORY       TAG       IMAGE ID       CREATED         SIZE
pacuna/webapp    915685c   b0a8963a7a68   4 minutes ago   850.6 MB
...
```

Thus I can see that my latest tag is `915685c`.

Open the `kube/jobs/setup-job.yaml` and change the tag for the container image.

```
containers:
- name: setup
  image: pacuna/webapp:915685c
```

The final template should look as follows:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: setup
spec:
  template:
    metadata:
      name: setup
    spec:
      containers:
      - name: setup
        image: pacuna/webapp:915685c
        command: ["/bin/bash", "./setup.production.sh"]
        env:
        - name: PASSENGER_APP_ENV
        value: production
    restartPolicy: Never
```

And now we can run the following template:

```
$ kubectl create -f kube/jobs/setup-job.yaml
```

Output:

```
job "setup" created
```

Check the logs after a few minutes to see if it's finished.

```
$ Pods=$(kubectl get Pods --selector=job-name=setup --output=jsonpath=
{.items..metadata.name})
$ kubectl logs $Pods
```

Output:

```
Waiting PostgreSQL to start on 5432...
PostgreSQL started
D, [2016-09-27T02:22:06.207320 #6] DEBUG -- : (21.4ms) CREATE TABLE "schema_
migrations" ("versio\
n" character varying PRIMARY KEY)
D, [2016-09-27T02:22:06.218350 #6] DEBUG -- : (8.0ms) CREATE TABLE "ar_
internal_metadata" ("key"\
 character varying PRIMARY KEY, "value" character varying, "created_at"
timestamp NOT NULL, "updated\
_at" timestamp NOT NULL)
D, [2016-09-27T02:22:06.219207 #6] DEBUG -- : (0.2ms) SELECT pg_try_
advisory_lock(38634698887904\
75145);
D, [2016-09-27T02:22:06.227247 #6] DEBUG -- : ActiveRecord::SchemaMigration
Load (0.4ms) SELECT "\
schema_migrations".* FROM "schema_migrations"
I, [2016-09-27T02:22:06.237053 #6] INFO -- : Migrating to CreateArticles
(20160925220117)
D, [2016-09-27T02:22:06.238575 #6] DEBUG -- : (0.1ms) BEGIN
== 20160925220117 CreateArticles: migrating ================================
-- create_table(:articles)
D, [2016-09-27T02:22:06.250670 #6] DEBUG -- : (10.9ms) CREATE TABLE
"articles" ("id" serial prim\
ary key, "title" character varying, "body" text, "created_at" timestamp NOT
NULL, "updated_at" times\
tamp NOT NULL)
   -> 0.0119s
== 20160925220117 CreateArticles: migrated (0.0121s) =======================

D, [2016-09-27T02:22:06.255595 #6] DEBUG -- : SQL (0.3ms) INSERT INTO
"schema_migrations" ("versi\
on") VALUES ($1) RETURNING "version" [["version", "20160925220117"]]
D, [2016-09-27T02:22:06.256840 #6] DEBUG -- : (1.0ms) COMMIT
D, [2016-09-27T02:22:06.260468 #6] DEBUG -- : ActiveRecord::InternalMetadata
Load (0.3ms) SELECT \
 "ar_internal_metadata".* FROM "ar_internal_metadata" WHERE "ar_internal_
metadata"."key" = $1 LIMIT \
$2 [["key", :environment], ["LIMIT", 1]]
D, [2016-09-27T02:22:06.263914 #6] DEBUG -- : (0.1ms) BEGIN
D, [2016-09-27T02:22:06.265326 #6] DEBUG -- : SQL (0.3ms) INSERT INTO "ar_
internal_metadata" ("ke\
y", "value", "created_at", "updated_at") VALUES ($1, $2, $3, $4) RETURNING
"key" [["key", "environm\
ent"], ["value", "production"], ["created_at", 2016-09-27 02:22:06 UTC],
["updated_at", 2016-09-27 0\
```

```
2:22:06 UTC]]
D, [2016-09-27T02:22:06.266304 #6] DEBUG -- : (0.7ms) COMMIT
D, [2016-09-27T02:22:06.266731 #6] DEBUG -- : (0.2ms) SELECT pg_advisory_
unlock(3863469888790475\
145)
```

We can see that the production database was correctly migrated.

Finally, for the web application we want to make the same changes we did for the setup job in regard to the `PASSENGER_APP_ENV` and tag for the image, plus a small change. Until now we have been using a `NodePort` type for the service. However, now that we are in production, we can use a `LoadBalancer` type, so we can have a static end point for that service. Open the `kube/deployments/webapp-deployment.yaml` file and change the service type to `LoadBalancer`.

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  ports:
    - port: 80
  selector:
    app: webapp
    tier: frontend
  type: LoadBalancer
```

Finally, change the tag image with your latest tag and also change the `PASSENGER_APP_ENV` to production. The final deployment template should look as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
  labels:
    app: webapp
  spec:
    ports:
      - port: 80
    selector:
      app: webapp
      tier: frontend
      type: LoadBalancer
    ---
    apiVersion: extensions/v1beta1
    kind: Deployment
    metadata:
```

```
      name: webapp
      labels:
        app: webapp
    spec:
      replicas: 3
      template:
      metadata:
        labels:
          app: webapp
          tier: frontend
      spec:
        containers:
          - image: pacuna/webapp:915685c
            name: webapp
            env:
            - name: PASSENGER_APP_ENV
              value: production
            ports:
            - containerPort: 80
              name: webapp
            imagePullPolicy: Always
```

Now we can create the webapp deployment:

```
$ kubectl create -f kube/deployments/webapp-deployment.yaml
service "webapp" created
deployment "webapp" created
```

If we inspect the service with the following:

```
$ kubectl describe service webapp
```

We'll see

```
Name:                   webapp
Namespace:              default
Labels:                 app=webapp
Selector:               app=webapp,tier=frontend
Type:                   LoadBalancer
IP:                     10.0.250.219
LoadBalancer Ingress: a333dae17845a11e6b47b06103f11903-585648094.us-west-2.
elb.amazonaws.com
Port:                   <unset>         80/TCP
NodePort:               <unset>         30426/TCP
Endpoints:              10.244.0.57:80,10.244.0.58:80,10.244.0.59:80
Session Affinity:       None
```

```
Events:
  FirstSeen  LastSeen  Count  From  SubobjectPath  Type
  ---------  --------  -----  ----  -------------  -------------------------
  32s        32s       1   {service-controller } Normal CreatingLoadBalancer
  30s        30s       1   {service-controller } Normal CreatedLoadBalancer
```

Kubernetes created an AWS Load Balancer for us and it's attached to this service. The end point is in the `LoadBalancer Ingress` field and my case corresponds to `a333dae17845a11e6b47b06103f11903-585648094.us-west-2.elb.amazonaws.com`.

We can also describe our deployment to see if our replicas are available.

```
$ kubectl describe deployment webapp
```

Output:

```
Name:                   webapp
Namespace:              default
CreationTimestamp:      Mon, 26 Sep 2016 23:29:22 -0300
Labels:                 app=webapp
Selector:               app=webapp,tier=frontend
Replicas:               3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          webapp-1367336656 (3/3 replicas created)
Events:
  FirstSeen  LastSeen   Count  From              SubobjectPath  Type
  ---------  --------   -----  ----              -------------  --------
  2m          2m         1     {deployment-controller } Normal  6656 to 3
```

There you see we have actually three pods running our application. The load balancer associated with the service will be balancing requests among these three replicas. This integration with AWS is pretty good. We just have to put the `LoadBalancer` type and Kubernetes automatically will create that resource in our account. Keep in mind that this kind of integration is possible with only a couple of cloud providers. Currently the best integrations are with Amazon AWS and Google GCE.

You'll have to wait for a few seconds for that end point to be alive. Once it's ready you can use the cURL post command to test it:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"my first
article","body":"Lorem ipsum dolor sit amet, consectetur adipiscing
elit..."}' http://a333dae17845a11e6b47b06103f11903-585648094.us-west-2.
elb.amazonaws.com/articles
```

Output:

```
{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit...","created_at":"2016-09-27T02:36:57.972Z",
"updated_at":"2016-09-27T02:36:57.972Z"}%
```

Great! The end point is working correctly.

We have successfully deployed our Rails API in a Kubernetes cluster. Now we are going to dig into more advanced topics such as Volumes for persisting data and running deployment updates to our application.

Adding Persistence

A very useful integration between Kubernetes and AWS is that you can mount AWS Volumes in a very transparent way into your containers. We only need to create a volume through the AWS API and then add a new section in the deployment file for the service you want to persist. In our case we only need persistence for the PostgreSQL container. With that in place, if the container crashes, the data will remain in the AWS Volume and can be read by the next container placed by Kubernetes.

Kubernetes currently has three recycling policies for Volumes. These are Retain, Recycle, and Delete. Depending on the type of Volume, you can use some of these policies. For example, for AWS you can use Delete, which will delete the associated storage (EBS) once the Volume is deleted.

Let's start by creating the volume. We can do this with a very simple command using the AWS CLI. The only detail here is that you need to use the same region your cluster lives in. If you didn't change the default one, then the region will be `us-west-2` and you can use something like `us-west-2a` for the availability zone. Remember that our default region may be different than the one Kubernetes used for creating the cluster, so let's also put it in the command along with the availability zone:

```
$ aws ec2 create-volume --region us-west-2 --availability-zone us-west-2a
--size 10 --volume-type gp2
```

Output:

```
{
    "AvailabilityZone": "us-west-2a",
    "Encrypted": false,
    "VolumeType": "gp2",
    "VolumeId": "vol-fe268f4a",
    "State":     "creating",
    "Iops": 100,
    "SnapshotId": "",
    "CreateTime": "2016-09-27T02:51:23.429Z",
    "Size": 10
}
```

The output gives us a `VolumeId` among several attributes for the created volume. We're going to need that Volume ID later for our template so don't lose it.

Now, let's open the deployment file for the PostgreSQL service and change the container specification so it looks as follows:

```
spec:
  containers:
  - image: postgres:9.5.3
    name: postgres
    env:
    - name: POSTGRES_PASSWORD
      value: mysecretpassword
    - name: POSTGRES_USER
      value: webapp
    - name: POSTGRES_DB
      value: webapp_production
    - name: PGDATA
      value: /var/lib/postgresql/data/pgdata
    ports:
    - containerPort: 5432
      name: postgres
      volumeMounts:
    - name: postgres-persistent-storage
      mountPath: /var/lib/postgresql/data
      volumes:
    - name: postgres-persistent-storage
      awsElasticBlockStore:
        volumeID: vol-fe268f4a
        fsType: ext4
```

Replace the `volumeID` property with the one you got previously.

In order to use the AWS volume, we have to declare a new `PGDATA` for the database and a volume, and also mount that volume using the `volumeMounts` property. Then, only by using its `VolumeId`, we can access the volume and save our data. As you can see, we are using the native `awsElasticBlockStore` property from Kubernetes which allows the integration with the EBS service from AWS. This kind of syntax for working with volumes is usual among container tools. First, you have a block that declares the properties of the volume, and then you mount that volume into your container using the `volumeMounts` in this case.

It's time to try out the volume. Let's delete our current deployment and launch a new one.

```
$ kubectl delete -f kube/deployments/postgres-deployment.yaml
```

And let's run it again with the new changes.

```
$ kubectl create -f kube/deployments/postgres-deployment.yaml
```

Output:

```
service "postgres" created
deployment "postgres" created
```

We can describe the postgres replica set to see if it is in fact using the volume.

```
$ kubectl describe rs postgres
```

Output:

```
Name:           postgres-560075503
Namespace:      default
Image(s):       postgres:9.5.3
Selector:       app=webapp,Pod-template-hash=560075503,tier=postgres
Labels:         app=webapp
                Pod-template-hash=560075503
                tier=postgres
Replicas:       1 current / 1 desired
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Volumes:
  postgres-persistent-storage:
    Type:       AWSElasticBlockStore (a Persistent Disk resource in AWS)
    VolumeID:   vol-fe268f4a
    FSType:     ext4
    Partition:  0
    ReadOnly: false
  Events:
  FirstSeen    LastSeen    Count    From       SubobjectPath    Type
  ---------    --------    -----    ----       -------------    --------
  28s          28s         1        {replicaset-controller }    Normal 5yl
```

There you can see the Volumes section showing that the volume is correctly mounted into the container.

We can also query the AWS API to see if can gives us some information about which instance this volume attached to it.

```
$ aws ec2 describe-volumes --volume-ids vol-fe268f4a --region us-west-2
```

Output:

```
{
    "Volumes": [
        {
            "AvailabilityZone": "us-west-2a",
            "Attachments": [
                {
                    "AttachTime": "2016-09-27T03:01:03.000Z",
                    "InstanceId": "i-8f86bb97",
                    "VolumeId": "vol-fe268f4a",
                    "State": "attached",
                    "DeleteOnTermination": false,
                    "Device": "/dev/xvdba"
                }
            ],
            "Encrypted": false,
            "VolumeType": "gp2",
            "VolumeId": "vol-fe268f4a",
            "State": "in-use",
            "Iops": 100,
            "SnapshotId": "",
            "CreateTime": "2016-09-27T02:51:23.429Z",
            "Size": 10
        }
    ]
}
```

There you can see that this volume is in fact attached to an instance. That's the instance that's holding the PostgreSQL container right now and belongs to the Kubernetes cluster.

If we want to do our smoke test one more time to see if the database is working correctly, we first have to rerun our setup job. Remember that this container migrates the database and right now we are starting with a fresh PostgreSQL container that from now on will be able to persist its data.

```
$ kubectl delete job/setup
$ kubectl create -f kube/jobs/setup-job.yaml
```

As always, let's check out the logs.

```
$ Pods=$(kubectl get Pods --selector=job-name=setup --output=jsonpath={.
items..metadata.name})
$ kubectl logs $Pods
```

Output:

```
Waiting PostgreSQL to start on 5432...
PostgreSQL started
D, [2016-09-27T03:06:16.395761 #6] DEBUG -- : (10.2ms) CREATE TABLE "schema_
migrations" ("versio\
n" character varying PRIMARY KEY)
D, [2016-09-27T03:06:16.406388 #6] DEBUG -- : (7.7ms) CREATE TABLE "ar_
internal_metadata" ("key"\
character varying PRIMARY KEY, "value" character varying, "created_at"
timestamp NOT NULL, "updated\
_at" timestamp NOT NULL)
D, [2016-09-27T03:06:16.407140 #6] DEBUG -- : (0.2ms) SELECT pg_try_
advisory_lock(38634698887904\
75145);
D, [2016-09-27T03:06:16.414507 #6] DEBUG -- : ActiveRecord::SchemaMigration
Load (0.3ms) SELECT "\
schema_migrations".* FROM "schema_migrations"
I, [2016-09-27T03:06:16.419156 #6] INFO -- : Migrating to CreateArticles
(20160925220117)
D, [2016-09-27T03:06:16.420605 #6] DEBUG -- : (0.1ms) BEGIN
== 20160925220117 CreateArticles: migrating ===================================
-- create_table(:articles)
D, [2016-09-27T03:06:16.429336 #6] DEBUG -- : (7.5ms) CREATE TABLE
"articles" ("id" serial prima\
ry key, "title" character varying, "body" text, "created_at" timestamp NOT
NULL, "updated_at" timest\
amp NOT NULL)
-> 0.0086s
== 20160925220117 CreateArticles: migrated (0.0087s) ==========================
D, [2016-09-27T03:06:16.433646 #6] DEBUG -- : SQL (0.3ms) INSERT INTO
"schema_migrations" ("versi\
on") VALUES ($1) RETURNING "version" [["version", "20160925220117"]]
D, [2016-09-27T03:06:16.435490 #6] DEBUG -- : (1.6ms) COMMIT
D, [2016-09-27T03:06:16.438876 #6] DEBUG -- : ActiveRecord::InternalMetadata
Load (0.3ms) SELECT \
"ar_internal_metadata".* FROM "ar_internal_metadata" WHERE "ar_internal_
metadata"."key" = $1 LIMIT \
$2 [["key", :environment], ["LIMIT", 1]]
D, [2016-09-27T03:06:16.442136 #6] DEBUG -- : (0.1ms) BEGIN
D, [2016-09-27T03:06:16.443314 #6] DEBUG -- : SQL (0.3ms) INSERT INTO "ar_
internal_metadata" ("ke\
y", "value", "created_at", "updated_at") VALUES ($1, $2, $3, $4) RETURNING
"key" [["key", "environm\
ent"], ["value", "production"], ["created_at", 2016-09-27 03:06:16 UTC],
["updated_at", 2016-09-27 0\
```

```
3:06:16 UTC]]
D, [2016-09-27T03:06:16.444709 #6] DEBUG -- : (1.1ms) COMMIT
D, [2016-09-27T03:06:16.445132 #6] DEBUG -- : (0.2ms) SELECT pg_advisory_
unlock(3863469888790475\
145)
```

We are good to go. Now let's run that test using our ELB DNS address.

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"my first
article","body":"Lorem ipsum dolor sit amet, consectetur adipiscing
elit..."}' http://a333dae17845a11e6b47b06103f11903-585648094.us-west-2.
elb.amazonaws.com/articles
```

Output:

```
{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit..\
.","created_at":"2016-09-27T03:07:36.706Z","updated_at":"2016-09-
27T03:07:36.706Z"}%
```

Great! It's working. But, how can we know that we are actually persisting the data? Well, there's only one way to know. Let's delete our PostgreSQL deployment and run it again one more time. This time we should not even need to run the setup, because the tables should still be there. Also, we have one article there, so let's hope it's still there after a new deployment.

```
$ kubectl delete -f kube/deployments/postgres-deployment.yaml
```

Output:

```
service "postgres" deleted
deployment "postgres" deleted
```

Now let's recreate it.

```
$ kubectl create -f kube/deployments/postgres-deployment.yaml
```

Output:

```
service "postgres" created
deployment "postgres" created
```

Let's check if we have a valid response and our first article back.

```
$ curl http://a333dae17845a11e6b47b06103f11903-585648094.us-west-2.elb.
amazonaws.com/articles
```

Output:

```
[{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit.\
..","created_at":"2016-09-27T03:07:36.706Z","updated_at":"2016-09-
27T03:07:36.706Z"}]%
```

Awesome! This means we can kill the PostgreSQL container and the data won't be lost. You can apply this trick for persisting any type of data (e.g., for a search engine like Elasticsearch or a different database like MySQL or MongoDB).

Updating the Application

One of the cool features of deployments in Kubernetes, is that you can run updates very easily. You only need to create the new version of the image you want to deploy, and then update the respective deployment templates. After that you can have a new version running with just one command.

We are going to add a new field to our articles table. This will require us to apply migrations, so we also have to run the setup container before the update. Don't worry, once we start to use Jenkins for our deployments, everything will be automatic. But for now let just stay with the kubectl commands and some bash scripts.

Let's add the new field in our Rails application.

```
$ docker-compose run --rm webapp bin/rails g migration AddSlugToArticles
slug:string
```

Let's also generate this slug automatically before the record gets saved in the `articles#create` action:

```
def create
  @article = Article.new(article_params)
  @article.slug = @article.title.parameterize

  if @article.save
    render json: @article, status: :created, location: @article
  else
    render json: @article.errors, status:    :unprocessable_entity
  end
end
```

OK, I'm pretty sure that's going to work. Let's rebuild the image and push it to DockerHub. Let's commit our changes so we have a new commit hash to tag the Docker image and then use our push script.

```
$ git add -A
$ git commit -m 'add slug field to articles'
$ ./push.sh
```

Now that we updated the image, we can change the tag in both the `setup-job` template and the `webapp-deployment` template to our latest commit hash.

```
$ git rev-parse --short HEAD
d15587c
```

And change the respective line for each file.

```
image: pacuna/webapp:d15587c
```

Now, let's rerun our setup job so the migrations are applied:

```
$ kubectl delete jobs/setup
$ kubectl create -f kube/jobs/setup-job.yaml
```

Check the logs.

```
$ Pods=$(kubectl get Pods --selector=job-name=setup --output=jsonpath={.
items..metadata.name})
$ kubectl logs $Pods
```

Output:

```
Waiting PostgreSQL to start on 5432...
PostgreSQL started
D, [2016-09-28T03:16:05.360263 #6] DEBUG -- : (1.0ms) SELECT pg_try_
advisory_lock(38634698887904\
75145);
D, [2016-09-28T03:16:05.369135 #6] DEBUG -- : ActiveRecord::SchemaMigration
Load (1.8ms) SELECT "\
schema_migrations".* FROM "schema_migrations"
I, [2016-09-28T03:16:05.374952 #6] INFO -- : Migrating to AddSlugToArticles
(20160928030155)
D, [2016-09-28T03:16:05.376440 #6] DEBUG -- : (0.1ms) BEGIN
== 20160928030155 AddSlugToArticles: migrating ==============================
-- add_column(:articles, :slug, :string)
D, [2016-09-28T03:16:05.385134 #6] DEBUG -- : (8.2ms) ALTER TABLE "articles"
ADD "slug" characte\
r varying
  -> 0.0085s
== 20160928030155 AddSlugToArticles: migrated (0.0087s) =====================

D, [2016-09-28T03:16:05.388399 #6] DEBUG -- : SQL (0.3ms) INSERT INTO
"schema_migrations" ("versi\
on") VALUES ($1) RETURNING "version" [["version", "20160928030155"]]
D, [2016-09-28T03:16:05.390099 #6] DEBUG -- : (1.4ms) COMMIT
D, [2016-09-28T03:16:05.395324 #6] DEBUG -- : ActiveRecord::InternalMetadata
Load (1.7ms) SELECT \
```

```
"ar_internal_metadata".* FROM "ar_internal_metadata" WHERE "ar_internal_
metadata"."key" = $1 LIMIT \
$2 [["key", :environment], ["LIMIT", 1]]
D, [2016-09-28T03:16:05.398654 #6] DEBUG -- : (0.1ms) BEGIN
D, [2016-09-28T03:16:05.399649 #6] DEBUG -- : (0.1ms) COMMIT
D, [2016-09-28T03:16:05.400064 #6] DEBUG -- : (0.2ms) SELECT pg_advisory_
unlock(3863469888790475\
145)
```

You can see the new migration ran correctly. And now, to update our deployment, we can run the following:

```
$ kubectl apply -f kube/deployments/webapp-deployment.yaml
```

Output:

```
service "webapp" configured deployment "webapp" configured
```

The `apply` command will check the differences between the current deployment and the passed template and it'll update the correspondent objects. In our case, the container changed so it has to update the pods. This command will be very helpful to automate this task, since we just need to replace the image tag with the new version and the run the command.

Wait for a second for the deployment to get updated and let's try to create a new article.

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"my second
article","body":"Lorem ipsum dolor sit amet, consectetur adipiscing
elit..."}' http://a333dae17845a11e6b47b06103f11903-585648094.us-west-2.
elb.amazonaws.com/articles
```

Output:

```
{"id":34,"title":"my second article","body":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit...","created_at":"2016-09-28T03:20:37.427Z",
"updated_at":"2016-09-28T03:20:37.427Z","slug":"my-second-article"}%
```

The slug was generated correctly, which means our migration was correctly applied and also the code added to the articles controller is working.

We just made our first real deployment to our application. This new concept of deployments allows us to run updates to our software very easily. You only need to generate a new image, and update your current deployment by keeping its metadata and then using the `apply` command. Then Kubernetes will update the pods and replica sets and it'll make sure the service can route the requests to the correct pods.

In the next section we'll see how to automate this entire process from your local environment. Although this could be a very good stopping point for most of the applications, we want to go further and build a solid continuous integration pipeline with Jenkins. But first, let's work on some scripts for automating the migrations and the deployments.

Automation Scripts

Automating the deployment is actually pretty easy. We just have to combine our current `push.sh` script with a new command that's going to set a new image in our current deployment. This command looks as follows:

```
$ kubectl set image deployment my-deployment container=image:new-tag
```

As you can see, we only need the name of our deployment, the container we want to update, and the new image the container should be built from. Normally this would be the same original image but with a different tag, in our case the latest commit hash.

Before deployments came into play, the usual way of updating services was by using the `rolling-update` command. This command will update one pod a time and it only supports replication controllers. Kubernetes recommends that you use deployments and replica sets to manage your updates instead of using replication controllers with rolling updates.

Let's create a new folder for this new deployment script and move the `push.sh` script inside it.

```
$ mkdir deploy
$ mv push.sh deploy
```

Now, edit the `deploy/push.sh` file and add the following, replacing it, of course, with your DockerHub repository:

```
#!/bin/sh
set -x

LC=$(git rev-parse --short HEAD)
docker build -f Dockerfile -t pacuna/webapp:${LC} .
docker push pacuna/webapp:${LC}
kubectl set image deployment webapp webapp=pacuna/webapp:${LC}
```

This is almost the same push script, but we added the `kubectl set` command to update our current deployment.

For the migrate job it's a little bit more complicated. Currently there's no easy way to update a job image and then rerun it. We first have to delete the current job, then change the image in the job template, and finally run it again. The first part is actually pretty easy; for deleting the job we can just use the following:

```
$ kubectl delete jobs/setup
```

Then for replacing the image with the latest tag, which we know is going to be the latest commit, we can add a placeholder instead of a real image tag in the file and then use a tool like **sed** to replace it with the real tag. Open the `kube/jobs/setup-job.yaml` and change the image to

```
image: pacuna/webapp:LAST_COMMIT
```

Then, with **sed**, we can find that placeholder, create a new file where the placeholder is replaced with the latest commit, run the `kubectl apply` command with that temp file, and finally delete it. Let's create the script and add those instructions.

```
$ touch deploy/migrate.sh
$ chmod +x deploy/migrate.sh
```

And add:

```
#!/bin/sh
set -x

# get latest commit hash
LC=$(git rev-parse --short HEAD)

# delete current migrate job
kubectl delete jobs/setup || true

# replace LAST_COMMIT with latest commit hash output the result to a tmp
file
sed "s/webapp:LAST_COMMIT/webapp:$LC/g" kube/jobs/setup-job.yaml > setup-
job.yaml.tmp

# run the updated tmp job in the cluster and then delete the file
kubectl create -f setup-job.yaml.tmp &&
  rm setup-job.yaml.tmp
```

And that's it! Now if you want to make changes to your code and deploy a new version, you just have to commit your changes and then run `deploy/push.sh`. If you want run migrations, you can just run `deploy/migrate.sh`. You can even build another script that uses both for every deployment, which is what we will later do with Jenkins.

Summary

After this chapter, you should know how to run a Rails application in a Kubernetes cluster. We started by studying the basics of the Kubernetes architecture and the main objects you'll use when running web applications: Pods, Replication Sets, Services, Volumes, Jobs, and Deployments. We talked about the importance of labels, which allow the match between the different objects that need to communicate. We were able to successfully deploy our example application and also add a persistence layer by using Volumes thanks to AWS EBS and the Kubernetes integration. Finally, we built two automation scripts to run fast deliveries. The first one was to run migrations and the second one was to run deployments to the actual source code.

Next, will see how to accomplish a similar workflow by using another big orchestration framework created by Amazon.

CHAPTER 4

Amazon EC2 Container Service

Concepts

Although every container orchestration framework uses its own specific language for its concepts, they all try to solve the same issues. That's why you'll find some similarities between Amazon EC2 Container Service (ECS) and Kubernetes but also a couple of differences in the way they solve the issues. One strong point in favor of ECS is that because it's a native AWS (Amazon Web Services) technology, it is completely integrated with the other components, such as VPC (Virtual Private Cloud), EC2, ELBs (Elastic Load Balancers), and Route53. The main issue with ECS is that it is coupled to AWS, and currently you can't run an ECS cluster on other cloud providers, which makes sense since it's an Amazon technology.

Let's review the main concepts in ECS.

Container Instance

A container instance is just a regular EC2 instance that is attached to the cluster. You can create a cluster using a tool called ECS CLI (Command Line Interface) and declare the number of instances you want, and they will be attached automatically. You can also use the graphical interface in the ECS panel and use a wizard to create a new cluster. Finally, if you prefer to create your instances by hand, you have to initialize them with some extra data that will be attached to the cluster. In this book we will be using the ECS CLI tool to create our cluster.

Task Definition

A task definition is the template where we declare a group of containers that'll run on the same instance. You'll probably will be running just one container per task definition, but it's good to know that you can run more than one container if you wish. Think of a task

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1_4

definition as a skeleton for a task, and the actual instances of the task definition will run within the cluster and will be managed by a task scheduler.

Following is an example task definition for our application:

```
{
    "family": "webapp",
    "containerDefinitions": [
        {
            "name": "webapp",
            "image": "pacuna/webapp:345opj",
            "cpu": 500,
            "memory": 500,
            "portMappings": [
                {
                    "containerPort": 80,
                    "hostPort": 80,
                    "protocol": "tcp"
                }
            ],
            "essential": true,
            "environment": [
                {
                    "name": "PASSENGER_APP_ENV",
                    "value": "production"
                }
            ]
        }
    ]
}
```

We start by declaring a family. This family, along with a revision number, will identify a task definition. Every time you make a change to the same task definition, you will create a new revision. Then, when you want to run a task, you can specify the family and revision number. If you don't, ECS will run the latest active revision. Then you have the container definitions. The elements are pretty similar to a Docker Compose declaration. We have to indicate the image, the ports, and the environmental variables we want to use. We also have to specify the requirements for this task (e.g., the CPU units and the memory limit for the container). This will help us to control the resources in our instances.

In order to run a task definition, you have to register it and then run it specifying the family and revision number. To register the task, you can use the following command:

```
$ aws ecs register-task-definition --cli-input-json file://webapp.json
```

We can directly pass the JSON (JavaScript Object Notation) file with the task definition. That will return the latest revision number assigned to it by ECS.

Then you can run the task with

```
$ aws ecs run-task --task-definition webapp:1 --cluster ecs-cluster
```

Indicating the family and revision number, along with the name of the cluster where you want to run it.

Another important aspect of tasks is that we can override a definition and run a task using a custom entry point. This will help us to run tasks that live in our application, like migrations or assets-related tasks.

We can declare a task override with

```
{
  "containerOverrides": [
    {
      "name": "webapp",
      "command": ["bin/rails", "db:migrate", "RAILS_ENV=production"],
      "environment": [
        {
          "name": "PASSENGER_APP_ENV",
          "value": "production"
        }
      ]
    }
  ]
}
```

And run it with

```
$ aws ecs run-task --task-definition webapp --overrides file://migrate-
overrides.json --cluster ecs-cluster
```

That's going to run a new task using the same definition we had for our main application, but with a custom entry point that's going to migrate the database. After that, the task will finish. This is similar to what we did in Kubernetes with the concept of Jobs.

Service

The service is an element that's going to give us a static end point for our running tasks. Normally it'll be associated with a load balancer so you can balance the traffic among your replicas. The service is also in charge of maintaining the desired number of running tasks. If you read about the concepts in Kubernetes, the ECS service is like a mix between a Replication Set and a Kubernetes Service.

If we want to associate an ELB with our service, we have to create it first. Then we can specify the ELB identifier in the service definition in the following way:

```
{
    "cluster": "ecs-cluster",
    "serviceName": "webapp-service",
    "taskDefinition": "webapp:1",
    "loadBalancers": [
```

```
        {
            "loadBalancerName": "webapp-load-balancer",
            "containerName": "webapp",
            "containerPort": 80
        }
    ],
    "desiredCount": 3,
    "role": "ecsServiceRole"
}
```

We need to specify a name for the service, the task definition we want to use to run the tasks, the ELB specifications, and the desired number of tasks that the service should keep running. We also have to specify the role that ECS created for our services.

Once we have this template, we can launch the service with the following:

```
$ aws ecs create-service --cli-input-json file://webapp-service.json
```

If later we create a new task definition revision and we want to update the associated service, we can run the following:

```
$ aws ecs update-service --service webapp-service --task-definition webapp
--desired-count 3 --cluster ecs-cluster
```

Since we didn't specify a revision number for the task, ECS will pick up the latest active revision and use that definition to run the tasks.

Configuring the ECS-CLI

Currently there are two ways to create an ECS cluster. You can use the graphical interface from the AWS panel for ECS or you can make use of the ESC CLI tool. We are going to focus on the latter for launching our cluster. Any of these choices will give you the same result. Keep in mind that it can be a little more tedious if you decide to configure a cluster by yourself. You'll need to launch instances with initial user data and security groups, among other configurations.

The ESC-CLI is pretty easy to install. You can find more details in the official documentation (http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_CLI_installation.html), but basically you'll need to run one command.

The version I'm currently running is

```
$ ecs-cli --version
```

Output:

```
ecs-cli version 0.4.4 (7e1376e)
```

If you have the AWS CLI tool already configured, meaning you have your AWS access keys configured for the client, you can make use of the ECS CLI tool immediately. If not,

you can follow the instructions provided at http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html#cli-quick-configuration.

Creating the Cluster Using the Amazon ECS CLI

Before we launch the cluster, we'll need a key pair to access the nodes. You can use an existent key or create and save a new one with the following command:

```
$ aws ec2 create-key-pair --key-name ecs-cluster-pem --query 'KeyMaterial'
--output text > ecs-cluster-pem.pem
```

Move that PEM to a safe place. You won't be able to download it again later.

Now we can use the `configure` command to configure and save our future cluster information. This command will create and save the configuration in a file so we don't have to specify all the data when running future commands. We only need to pass the name we want for our cluster.

```
$ ecs-cli configure --cluster ecs-cluster --region us-east-1
```

Output:

```
INFO[0000] Saved ECS CLI configuration for cluster (ecs-cluster)
```

The configuration file is located at `~/.ecs/config`. If you inspect that file, you'll see that none of our AWS credentials are actually there. You could also have passed those tokens with the configure commands, and they would have been saved in the file. But since we already have our credentials configured to use the AWS CLI, the ECS CLI knows where to find the credentials when they aren't passed as arguments.

Now that we have the configuration ready for the cluster, we can launch it along with some nodes. Let's use two `t2.medium` nodes. We only need to pass the key pair we created previously and the number and size of the nodes. ECS CLI will use the configuration saved to get our cluster information.

```
$ ecs-cli up --keypair ecs-cluster-pem --capability-iam --size 2 --instance-
type t2.medium
```

Output:

```
INFO[0002] Created cluster                    cluster=ecs-cluster
INFO[0003] Waiting for your cluster resources to be created
INFO[0004] Cloudformation stack status        stackStatus=CREATE_IN_PROGRESS
INFO[0066] Cloudformation stack               status stackStatus=CREATE_IN_
PROGRESS
INFO[0128] Cloudformation stack status        stackStatus=CREATE_IN_PROGRESS
INFO[0190] Cloudformation stack status        stackStatus=CREATE_IN_PROGRESS
INFO[0251] Cloudformation stack status        stackStatus=CREATE_IN_PROGRESS
```

This can take a while, and even after the command finishes, you'll have to wait for the instances to be initialized so they can join the cluster. Besides launching nodes for the cluster, this command will create a VPC and all the security resources for keeping our cluster safe.

Let's use the AWS CLI to ask about the clusters in our account.

```
$ aws ecs --region us-east-1 describe-clusters
```

Output:

```
{
    "failures": [],
    "clusters": [
        {
            "pendingTasksCount": 0,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/
            default",
            "status": "INACTIVE",
            "activeServicesCount": 0,
            "registeredContainerInstancesCount": 0,
            "clusterName": "default",
            "runningTasksCount": 0
        }
    ]
}
```

There you can see that the cluster we just launched was actually created, but it doesn't have any instances registered yet. That's because these are still initializing.

Wait for a few minutes, and let's run another command in order to get information about that specific cluster we just launched.

```
$ aws ecs --region us-east-1 describe-clusters --clusters ecs-cluster
```

Output:

```
{
    "clusters": [
        {
            "status": "ACTIVE",
            "clusterName": "ecs-cluster",
            "registeredContainerInstancesCount": 2,
            "pendingTasksCount": 0,
            "runningTasksCount": 0,
            "activeServicesCount": 0,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/
            ecs-cluster"
        }
    ],
    "failures": []
}
```

OK, that's better. Now you can see that the cluster has a status of active and it has two instances registered. You can also see more information about the tasks and services in our cluster. Don't worry about that yet. We'll see that later when we start to run containers in the cluster.

DB Configuration

AWS ECS doesn't currently have native support for DNS (Domain Name System) discovery like Kubernetes has. This is a big issue when you want to run a lot of different services in the cluster and even bigger if lots of those services need to be non-public. This is because the typical form for service discovery in ECS is done via load balancing. So every time you create a service that you need to be discoverable by other services, you need to create a load balancer for it. This can be expensive if you're running a micro-service architecture. In Kubernetes we solve this problem via DNS discovery. You can launch services like databases without exposing them to the Internet. They will have an internal IP (Internet Protocol) address and also an alias so the other services can communicate with them.

Another problem with ECS is that there's no support for associating cloud storage with the instances. You can mount volumes from the containers to the instances, but that doesn't work if your container will be jumping from one instance to another, which is what typically occurs during deployments. Again, Kubernetes has support for persistence by integrating itself with cloud provider native storage objects, like EBS (Elastic Block Storage) in the case of AWS. That storage will always be external to the cluster and the nodes, so it doesn't matter if a new container is launched on a different node than it was before, because the data lives outside the cluster.

These two reasons (no native support for DNS discovery and no support for storage objects) are why I recommend that you don't use ECS to run containers that need persistence. This goes for things like databases, search engines, cache storages, and so on.

Fortunately for us, we don't have to mount a server with a database engine and configure the whole thing by ourselves, since AWS has a very good database-as-a-service software called RDS. This service will allow us to run a new database server already configured and which we can use to connect to our web application that will be running on a container.

Creating a RDS Resource

The only complexity in creating a RDS resource is that we have to make sure to create it inside the same VPC that the ECS CLI created for our cluster. That's a much more secure approach than just to launch a server open to the Internet. We want to run a database that's only accessible from the same VPC.

First, let's collect some data that we may need during the execution of the following commands. These are our VPC ID and the ID of one of the VPC's subnets.

Let's describe all of our current VPC IDs and the first tag.

```
$ aws ec2 describe-vpcs --region us-east-1 --query="Vpcs[*].
{ID:VpcId,tags:Tags[0]}"
```

Output:

```
[
    {
        "ID": "vpc-e61cec82",
        "tags": null
    },
    {
        "ID": "vpc-a0e9c0c7",
        "tags": {
            "Value": "amazon-ecs-cli-setup-ecs-cluster",
            "Key": "aws:cloudformation:stack-name"
        }
    }
]
```

You can see I have two VPCs and obviously the second one is the one that was created for our cluster. It has a tag with a value that contains the name of the cluster. The ID for this VPC is `vpc-a0e9c0c7`. Remember to save yours.

Now we can get the subnet IDs for this VPC with the following command:

```
$ aws ec2 describe-subnets --filters "Name=vpc-id,Values=vpc-a0e9c0c7"
--query="Subnets[*].SubnetId"
```

Output:

```
[
    "subnet-3a09f717",
    "subnet-e0906cbb"
]
```

You should also save the values you got for your VPC.

Finally, we also want the security group ID that ECS CLI assigned to our VPC. We want to launch the database (DB) server with this same security group so the communication can be configured more easily. Let's get our security groups and filter by our VPC ID.

```
$ aws ec2 describe-security-groups --filters="Name=vpc-id,Values=vpc-
a0e9c0c7" --query="SecurityGroups[*].{Description:Description,ID:GroupId}"
```

Output:

```
[
    {
        "Description": "ECS Allowed Ports",
        "ID": "sg-bbe6b3c1"
    },
```

```
    {
        "Description": "default VPC security group",
        "ID": "sg-efe6b395"
    }
]
```

Right now I have two security groups for that VPC, but you can see one that says "ECS Allowed Ports." That's the one I want to use for the DB. Save that security group ID for our following command.

We need all these identifiers to create our RDS in our VPC. In fact, before launching the server, we have to create a DB Subnet Group. This group is just a collection of subnets designated for the RDS DB instance in our VPC. We got our subnet IDs because they're necessary to create this DB subnet group.

In the following command, replace the subnet IDs with the ones you got previously. You can leave the rest of argument just like that.

```
$ aws rds create-db-subnet-group --db-subnet-group-name webapp-postgres-
subnet --subnet-ids subnet-3a09f717 subnet-e0906cbb --db-subnet-group-
description "Subnet for PostgreSQL" \
```

Output:

```
{
    "DBSubnetGroup": {
        "Subnets": [
            {
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-3a09f717",
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1c"
                }
            },
            {
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-e0906cbb",
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1a"
                }
            }
        ],
        "VpcId": "vpc-a0e9c0c7",
        "DBSubnetGroupDescription": "Subnet for PostgreSQL",
        "SubnetGroupStatus": "Complete",
        "DBSubnetGroupArn": "arn:aws:rds:us-east-
        1:586421825777:subgrp:webapp-postgres-subnet",
        "DBSubnetGroupName": "webapp-postgres-subnet"
    }
}
```

The following command will create a DB server for us. Here we are going to create a database named `webapp_production`, with a master username `webapp` and a password `mysecretpassword`. We are also indicating the size of the server and the size of the associated storage. We also have to indicate the engine we want for our database. In our case we are using postgres. Finally, we are passing the identifier of the previously created DB subnet group and the VPC security group ID so this server is created in the same VPC where our cluster is.

```
$ aws rds create-db-instance --db-name webapp_production --db-instance-
identifier webapp-postgres \
--allocated-storage 20 --db-instance-class db.t2.medium --engine postgres \
--master-username webapp --master-user-password mysecretpassword --db-
subnet-group-name webapp-postg\ res-subnet \
--vpc-security-group-id sg-bbe6b3c1
```

Output:

```
{
    "DBInstance": {
        "PubliclyAccessible": false,
        "MasterUsername": "webapp",
        "MonitoringInterval": 0,
        "LicenseModel": "postgresql-license",
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-bbe6b3c1"
            }
        ],
        "CopyTagsToSnapshot": false,
        "OptionGroupMemberships": [
            {
                "Status": "in-sync",
                "OptionGroupName": "default:postgres-9-5"
            }
        ],
        "PendingModifiedValues": {
            "MasterUserPassword": "****"
        },
        "Engine": "postgres",
        "MultiAZ": false,
        "DBSecurityGroups": [],
        "DBParameterGroups": [
            {
                "DBParameterGroupName": "default.postgres9.5",
                "ParameterApplyStatus": "in-sync"
            }
        ],
```

```
        "AutoMinorVersionUpgrade": true,
        "PreferredBackupWindow": "04:05-04:35",
        "DBSubnetGroup": {
            "Subnets": [
                {
                    "SubnetStatus": "Active",
                    "SubnetIdentifier": "subnet-3a09f717",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1c"
                    }
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetIdentifier": "subnet-e0906cbb",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1a"
                    }
                }
            ],
            "DBSubnetGroupName": "webapp-postgres-subnet",
            "VpcId": "vpc-a0e9c0c7",
            "DBSubnetGroupDescription": "Subnet for PostgreSQL",
            "SubnetGroupStatus": "Complete"
        },
        "ReadReplicaDBInstanceIdentifiers": [], "AllocatedStorage": 20,
        "DBInstanceArn": "arn:aws:rds:us-east-1:586421825777:db:webapp-
        postgres",
        "BackupRetentionPeriod": 1,
        "DBName": "webapp_production",
        "PreferredMaintenanceWindow": "tue:06:28-tue:06:58",
        "DBInstanceStatus": "creating",
        "EngineVersion": "9.5.2",
        "DomainMemberships": [],
        "StorageType": "standard",
        "DbiResourceId": "db-3QR6VRBU4OIDKYCA73XFNINCH4",
        "CACertificateIdentifier": "rds-ca-2015",
        "StorageEncrypted": false,
        "DBInstanceClass": "db.t2.medium",
        "DbInstancePort": 0,
        "DBInstanceIdentifier": "webapp-postgres"
    }
}
```

You can query the API to get the current DB instance status with the following command:

```
$ aws rds describe-db-instances --db-instance-identifier webapp-postgres
--query 'DBInstances[*].{Status:DBInstanceStatus}'
[
    {
        "Status": "creating"
    }
]
```

Wait for a few minutes until the DB is ready and the status becomes available.

```
[
     {
        "Status": "available"
     }
]
```

Once the server is available, we can query the end point AWS gave to the DB:

```
$ aws rds describe-db-instances --db-instance-identifier webapp-postgres
--query 'DBInstances[*].{URL:Endpoint.Address}'
```

Output:

```
[
    {
        "URL": "webapp-postgres.caxygd3nh0bk.us-east-1.rds.amazonaws.com"
    }
]
```

Great! Now we have our database ready. But before we configure it in our application we have to do one thing. Right now we are using the cluster's VPC security group. Currently this group should only have an inbound rule for the port 80. What we want is to allow all traffic between elements that live inside this VPC.

We can create a custom rule for this with the following command. Remember to change the ID with the security group ID you got previously:

```
$ aws ec2 authorize-security-group-ingress --group-id sg-bbe6b3c1 --protocol
all --port all --source-group sg-bbe6b3c1
```

Let's also add a rule for accessing the nodes via ssh from anywhere. This can be useful for debugging and diagnosing container errors.

```
$ aws ec2 authorize-security-group-ingress --group-id sg-bbe6b3c1 --protocol
tcp --port 22 --cidr 0.0.0.0/0
```

Now our nodes should be able to access this database. You won't be able to connect to this database from outside the cluster. That could be an inconvenience if you like to inspect your databases using a graphical user interface (GUI), but it's also a very strong security measure to prevent attacks.

Now we can configure the production database credentials in our application.

```
production:
  <<: *default
  host: webapp-postgres.caxygd3nhObk.us-east-1.rds.amazonaws.com
  database: webapp_production
  username: webapp
  password: mysecretpassword
```

Let's rebuild our Docker image and add a tag for ECS using our latest commit

```
$ git add .
$ git commit -m 'add database credentials for rds'
$ LC=$(git rev-parse --short HEAD)
$ docker build -t pacuna/webapp:ecs-${LC} .
```

And push it to DockerHub.

```
$ docker push pacuna/webapp:ecs-${LC}
```

And that's it for the configuration. Now it's time to build the templates to deploy the application in the cluster.

Creating the Task Definition

If you follow the instructions to deploy with Kubernetes, you saw that we created an independent kube folder to keep the Kubernetes templates. Now let's create a folder to keep the ECS templates. Run the following commands in the root of the project:

```
$ mkdir ecs
$ mkdir ecs/task-definitions
```

Now we can create a skeleton task definition for our application with the following command:

```
$ aws ecs register-task-definition --generate-cli-skeleton > ecs/task-
definitions/webapp.json
```

That's going to create an empty task definition which we can fill out with our data.

Let's clean up the file a little bit and add the information for our first task definition.

Open the file and replace its content with the following:

```
{
    "family": "webapp",
    "containerDefinitions": [
        {
            "name": "webapp",
            "image": "pacuna/webapp:ecs-0eddbb1",
            "cpu": 500,
            "memory": 500,
            "portMappings": [
                {
                    "containerPort": 80,
                    "hostPort": 80,
                    "protocol": "tcp"
                }
            ],
            "essential": true,
            "environment": [
                {
                    "name": "PASSENGER_APP_ENV",
                    "value": "production"
                }
            ]
        }
    ]
}
```

For the image, you should use the Docker image we created after we set up the database credentials. It should be `ecs-` followed by the last commit.

Now we can register this file with

```
$ aws ecs register-task-definition --cli-input-json file://ecs/task-
definitions/webapp.json
```

Output:

```
{
        "taskDefinition": {
            "status": "ACTIVE",
            "family": "webapp",
            "volumes": [],
            "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:
            task-definition/webapp:1",
            "containerDefinitions": [
```

```
                {
                    "environment": [
                    {
                        "name": "PASSENGER_APP_ENV",
                        "value": "production"
                    }
                ],
                "name": "webapp",
                "mountPoints": [],
                "image": "pacuna/webapp:ecs-0eddbb1",
                "cpu": 500,
                "portMappings": [
                    {
                        "protocol": "tcp",
                        "containerPort": 80,
                        "hostPort": 80
                    }
                ],
                "memory": 500,
                "essential": true,
                "volumesFrom": []
            }
        ],
        "revision": 1
    }
}
```

The important part of the output is the family name we declared in the JSON file and the revision number. Every time we modify a task definition we create a new revision. In this case I got revision 1 because it's the first time I registered this task.

We can run this task with

```
$ aws ecs run-task --task-definition webapp:1 --cluster ecs-cluster
```

Output:

```
{
    "failures": [],
    "tasks": [
        {
            "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-
            811c-4672-ab95-2c343930825c\ ",
            "overrides": {
                "containerOverrides": [
                    {
                        "name": "webapp"
                    }
                ]
            },
```

```
            "lastStatus": "PENDING",
            "containerInstanceArn": "arn:aws:ecs:us-east-
            1:586421825777:container-instance/edd5e9a7-\ 7f29-43a6-98c1-
            b7d7dd912cd2",
            "createdAt": 1475113475.222,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/ecs-
            cluster", "desiredStatus": "RUNNING",
            "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:task-
            definition/webapp:1",
            "containers": [
                {
                    "containerArn": "arn:aws:ecs:us-east-
                    1:586421825777:container/2d4aa3c7-48d7-40c1\
                    -9b30-8938566c2b0c",
                    "taskArn": "arn:aws:ecs:us-east-
                    1:586421825777:task/1133a3a7-811c-4672-ab95-2c34\
                    3930825c"
                    "lastStatus": "PENDING",
                    "name": "webapp"
                }
            ]
        }
    ]
}
```

That command is going to run the container declared in the task definition somewhere in the cluster. If we want to get more info about our task, first we have to get its identifier.

```
$ aws ecs list-tasks --cluster ecs-cluster
```

Output:

```
{
    "taskArns": [
        "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-811c-4672-ab95-
        2c343930825c"
    ]
}
```

Now we can use that identifier in the order we want.

```
$ aws ecs describe-tasks --tasks arn:aws:ecs:us-east-
1:586421825777:task/1133a3a7-811c-4672-ab95-2c3\
43930825c --cluster ecs-cluster
```

Output:


```
{
    "failures": [],
    "tasks": [
        {
            "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-
            811c-4672-ab95-2c343930825c\ ",
            "overrides": {
                "containerOverrides": [
                    {
                        "name": "webapp"
                    }
                ]
            },
            "lastStatus": "RUNNING",
            "containerInstanceArn": "arn:aws:ecs:us-east-
            1:586421825777:container-instance/edd5e9a7-\
            7f29-43a6-98c1-b7d7dd912cd2",
            "createdAt": 1475113475.222,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/ecs-
            cluster",
            "startedAt": 1475113521.285,
            "desiredStatus": "RUNNING",
            "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:task-
            definition/webapp:1",
            "containers": [
                {
                    "containerArn": "arn:aws:ecs:us-east-
                    1:586421825777:container/2d4aa3c7-48d7-40c1\
                    -9b30-8938566c2b0c",
                    "taskArn": "arn:aws:ecs:us-east-
                    1:586421825777:task/1133a3a7-811c-4672-ab95-2c34\
                    3930825c",
                    "lastStatus": "RUNNING",
                    "name": "webapp",
                    "networkBindings": [
                        {
                            "protocol": "tcp",
                            "bindIP": "0.0.0.0",
                            "containerPort": 80,
                            "hostPort": 80
                        }
                    ]
                }
            ]
        }
    ]
}
```

We can see that the last status was running. If in your case it says pending, just wait until the image gets pulled and the container starts. Now, we know our application is not going to work out of the box. We have our database created but we haven't run the migrations yet. In Kubernetes we use the concept of Jobs for running this kind of stuff. In this case we are going to run a task that's going to override a task definition. This means we can run this same task but override its entry point, which is what we need.

Let's create a new JSON file that's going to contain the overrides for the migrations,

```
$ touch ecs/task-definitions/migrate-overrides.json
```

And add the following to it:

```
{
  "containerOverrides": [
    {
      "name": "webapp",
      "command": ["bin/rails", "db:migrate", "RAILS_ENV=production"],
      "environment": [
        {
          "name": "PASSENGER_APP_ENV",
          "value": "production"
        }
      ]
    }
  ]
}
```

As you can see, there's nothing crazy going on there. We are just setting a new entry point and making sure the environmental variables are preserved.

In order to run a task with this overrides we can run the following command:

```
$ aws ecs run-task --task-definition webapp:1 --overrides file://ecs/task-
definitions/migrate-overrides.json --cluster ecs-cluster
```

Output:

```
{
    "failures": [],
    "tasks": [
        {
            "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/865d7bcc-
            55c3-4c2b-993f-4bbd32c1f63f\ ",
            "overrides": {
                "containerOverrides": [
                    {
                        "environment": [
                            {
                                "name": "PASSENGER_APP_ENV",
```

```
                            "value": "production"
                        }
                    ],
                    "command": [
                        "bin/rails",
                        "db:migrate",
                        "RAILS_ENV=production"
                    ],
                    "name": "webapp"
                }
            ]
        },
        "lastStatus": "PENDING",
        "containerInstanceArn": "arn:aws:ecs:us-east-
        1:586421825777:container-instance/931096f1-\
        4e01-4baf-8cfc-73a80e5b1ead",
        "createdAt": 1475113727.013,
        "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/
        ecs-cluster",
        "desiredStatus": "RUNNING",
        "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:task-
        definition/webapp:1",
        "containers": [
            {

                "containerArn": "arn:aws:ecs:us-east-1:586421825777:cont
                ainer/2b547545-1bed-483b\
                -bb4a-dbc73e9de3ad",
                "taskArn": "arn:aws:ecs:us-east-
                1:586421825777:task/865d7bcc-55c3-4c2b-993f-4bbd\
                32c1f63f",
                "lastStatus": "PENDING",
                "name": "webapp"
            }
        ]
    }
]
}
```


That's going to take our same task definition, override its entry point so the migrations are executed, and then stop the task.

Now we just need to get the IP address of the node that's running our application. This can be a little bit tedious to do using the AWS CLI, but don't worry. Later we'll create a service with an associated load balancer that's going to give us a static DNS for our container.

Following are the steps to get the IP address of the node that's running the task:

1. Get our task identifier:

```
$ aws ecs list-tasks --cluster ecs-cluster

{
    "taskArns": [
        "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-811c-4672-ab95-
        2c343930825c"
    ]
}
```

2. Describe the task with that identifier filtered by the container instance identifier:

```
$ aws ecs describe-tasks --tasks arn:aws:ecs:us-east-
1:586421825777:task/1133a3a7-811c-4672-ab95-2c3\
43930825c --cluster ecs-cluster --query="tasks[*].containerInstanceArn"

[
    "arn:aws:ecs:us-east-1:586421825777:container-instance/edd5e9a7-7f29-
    43a6-98c1-b7d7dd912cd2"
]
```

3. Using the instance identifier, get the instance ID:

```
$ aws ecs describe-container-instances --container-instances arn:aws:ecs:us-
east-1:586421825777:container-instance/edd5e9a7-7f29-43a6-98c1-b7d7dd912cd2
--query="containerInstances[*].ec2InstanceId" --cluster ecs-cluster

[
    "i-69f4a17f"
]
```

4. Using that ID, get the IP address by using the EC2 API:

```
$ aws ec2 describe-instances --instance-ids i-69f4a17f
--query="Reservations[0].Instances[0].PublicIpAddress"

"54.164.16.149"
```

Now we know that the container is mapping its port 80 to the port 80 of the host, so we can cURL that IP address in one of our end points to see if it's actually responding.

```
$ curl -I http://54.164.16.149/articles

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Status: 200 OK
Cache-Control: max-age=0, private, must-revalidate
ETag: W/"4f53cda18c2baa0c0354bb5f9a3ecbe5"
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Runtime: 0.011589
X-Request-Id: 049c50bd-4f6e-4170-956c-8a085750edb8
Date: Thu, 29 Sep 2016 01:51:54 GMT
X-Powered-By: Phusion Passenger 5.0.29
Server: nginx/1.10.1 + Phusion Passenger 5.0.29
```

Great! Our application is up and running with no errors.

In the next section we'll create a service that's going to keep our task alive 24/7 and also create a load balancer so we can have a static address for our application.

Creating a Service for Our Application

As I mentioned previously, one of the cool features of ECS Services is that you can attach a load balancer to them. This will allow us to update our service and task definitions during deployments while keeping the address of the service.

One important thing is that we need a path where the load balancer can check if the container to which the requests are being routed is responding correctly. By default, the load balancer will hit the root path of our application. Since we don't have any action taking care of that route, production is going to respond with a not found status code, which will produce a fail with the health check.

Let's fix that by quickly adding an action that responds with a 200 status code for that path. In the routes file add the following route:

```
get '/', to: "pages#welcome"
```

Create a new controller

```
$ touch app/controllers/pages_controller.rb
```

And add the following to it:

```
class PagesController < ApplicationController
  def welcome
    render json: {message: 'hello!'}, status: 200
  end
end
```

And that's it. Now our base path will respond with a 200 status code if the container starts correctly and the load balancer will be attached with no problems.

We can create a load balancer in the VPC with (replace with your cluster's security group ID and a subnet ID in your VPC):

```
$ aws elb create-load-balancer --load-balancer-name webapp-load-balancer
--listeners "Protocol=HTTP,LoadBalancerPort=80,InstanceProtocol=HTTP,
InstancePort=80" --subnets subnet-3a09f717 subnet-e0906cbb--security-groups
sg-bbe6b3c1
```

Output:

```
{
     "DNSName": "webapp-load-balancer-1711291190.us-east-1.elb.amazonaws.com"
}
```

Now we can use that load balancer when creating the service.

First, let's stop the task that's running our application. For that, we'll need its identifier.

```
$ aws ecs list-tasks --cluster ecs-cluster
```

Output:

```
{
    "taskArns": [
        "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-811c-4672-ab95-
        2c343930825c"
    ]
}
```

And now we can run the stop-task command.

```
$ aws ecs stop-task --task arn:aws:ecs:us-east-1:586421825777:task/
1133a3a7-811c-4672-ab95-2c343930825c --cluster ecs-cluster
```

Output:


```
{
    "task": {
        "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/1133a3a7-811c-
        4672-ab95-2c343930825c",
        "overrides": {
            "containerOverrides": [
                {
                    "name": "webapp"
                }
            ]
        },
        "lastStatus": "RUNNING",
        "containerInstanceArn": "arn:aws:ecs:us-east-
        1:586421825777:container-instance/edd5e9a7-7f29\
        -43a6-98c1-b7d7dd912cd2",
        "createdAt": 1475113475.222,
        "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/
        ecs-cluster",
        "startedAt": 1475113521.285,
        "desiredStatus": "STOPPED",
        "stoppedReason": "Task stopped by user",
        "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:task-
        definition/webapp:1",
        "containers": [
            {
                "containerArn": "arn:aws:ecs:us-east-
                1:586421825777:container/2d4aa3c7-48d7-40c1-9b3\
                0-8938566c2b0c",
                "taskArn": "arn:aws:ecs:us-east-
                1:586421825777:task/1133a3a7-811c-4672-ab95-2c343930\825c",
                "lastStatus": "RUNNING",
                "name": "webapp",
                "networkBindings": [
                    {
                        "protocol": "tcp",
                        "bindIP": "0.0.0.0",
                        "containerPort": 80,
                        "hostPort": 80
                    }
                ]
            }
        ]
    }
}
```

If we list the tasks again, we shouldn't see anything.

```
$ aws ecs list-tasks --cluster ecs-cluster
```

Output:

```
{
    "taskArns": []
}
```

Let's create a folder for our services.

```
$ mkdir ecs/services
```

And now let's use the following command to generate a service skeleton:

```
$ aws ecs create-service --generate-cli-skeleton > ecs/services/webapp-
service.json --cluster ecs-cluster
```

Open the generated file and replace the values with the following:

```
{
    "cluster": "ecs-cluster",
    "serviceName": "webapp-service",
    "taskDefinition": "webapp:1",
    "loadBalancers": [
        {
            "loadBalancerName": "webapp-load-balancer",
            "containerName": "webapp",
            "containerPort": 80
        }
    ],
    "desiredCount": 1,
    "role": "ecsServiceRole"
}
```

And then create the service with

```
$ aws ecs create-service --cli-input-json file://ecs/services/webapp-
service.json
```

Output:

```
{
    "service": {
        "status": "ACTIVE",
        "taskDefinition": "arn:aws:ecs:us-east-1:586421825777:task-
        definition/webapp:1",
```

```
        "pendingCount": 0,
        "loadBalancers": [
            {
                "containerName": "webapp",
                "containerPort": 80,
                "loadBalancerName": "webapp-load-balancer"
            }
        ],
        "roleArn": "arn:aws:iam::586421825777:role/ecsServiceRole",
        "createdAt": 1475114557.793,
        "desiredCount": 2,
        "serviceName": "webapp-service",
        "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/ecs-
        cluster",
        "serviceArn": "arn:aws:ecs:us-east-1:586421825777:service/webapp-
        service",
        "deploymentConfiguration": {
            "maximumPercent": 200,
            "minimumHealthyPercent": 100
        },
        "deployments": [
             {
                "status": "PRIMARY",
                "pendingCount": 0,
                "createdAt": 1475114557.793,
                "desiredCount": 1,
                "taskDefinition": "arn:aws:ecs:us-east-1:586421825777:task-
                definition/webapp:1",
                "updatedAt": 1475114557.793,
                "id": "ecs-svc/9223370561740218014",
                "runningCount": 0
            }
        ],
        "events": [],
        "runningCount": 0
    }
}
```

Now we can try to hit the ELB URL (uniform resource locator) to see if it's actually routing requests to our application. Remember that we got the DNS when we called the following create load balancer command:

```
$ curl -I webapp-load-balancer-1711291190.us-east-1.elb.amazonaws.com/
articles

HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Content-Type: application/json; charset=utf-8
```

```
Date: Thu, 29 Sep 2016 02:03:33 GMT
ETag: W/"4f53cda18c2baa0c0354bb5f9a3ecbe5"
Server: nginx/1.10.1 + Phusion Passenger 5.0.29
Status: 200 OK
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Powered-By: Phusion Passenger 5.0.29
X-Request-Id: 5e670c6e-dde7-48bd-bb61-00d5598946b4
X-Runtime: 0.012451
X-XSS-Protection: 1; mode=block
Connection: keep-alive
```

The cool thing about the ELB is that it is going to balance the load among the number of tasks you define in your desired replicas. It wouldn't be able to do it without a service. In our case we're using just one replica, since we only have two nodes available. The problem is that this type of ELB won't allow us to have two containers exposing the same port on the same instance. And every time we update a service, a new container starts in the cluster before the old one dies. So during the deployment, there will be always a moment where two containers will be running and exposing the same port. In a real scenario where you need high availability for your application, you'll want to have a cluster with many more nodes and create services that keep a higher number of replicas running at the same time.

Let's try to create an article to see if everything is working correctly.

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"the
title","body":"The body"}' http\
://webapp-load-balancer-1711291190.us-east-1.elb.amazonaws.com/articles
```

Output:

```
{"id":1,"title":"the title","body":"The body","created_at":"2016-09-
29T02:04:32.267Z","updated_at":"\
2016-09-29T02:04:32.267Z","slug":"the-title"}%
```

Cool! So we have our DB and our load balancer working correctly.

In the next section we will see how to run updates to our application and how we can automate those updates with a little bit of bash scripting.

Running Updates to Our Application

Just as we did with Kubernetes, we want to use a couple of automated scripts to run updates for our application. We have to be able to automate two things: migrating our database in case there are new migrations and, after building a new version of our image, creating a new task definition revision with that new tag and updating our service to use that new task definition.

First, let's create a new deploy folder inside the ecs folder to keep these scripts and two empty files: one for pushing and one to migrate.

```
$ mkdir ecs/deploy
$ touch ecs/deploy/push.sh
$ touch ecs/deploy/migrate.sh
$ chmod +x ecs/deploy/*
```

Add the following to the push.sh file:

```
#!/bin/sh
set -x

LC=$(git rev-parse --short HEAD)
docker build -f Dockerfile -t pacuna/webapp:${LC} .
docker push pacuna/webapp:${LC}

# replace LAST_COMMIT with latest commit hash output the result to a tmp
file
sed "s/webapp:LAST_COMMIT/webapp:$LC/g" ecs/task-definitions/webapp.json >
webapp.json.tmp

# register the new task definition and delete the tmp file
aws ecs register-task-definition --cli-input-json file://webapp.json.tmp rm
webapp.json.tmp

# update the service
aws ecs update-service --service webapp-service --task-definition webapp
--desired-count 1 --cluster ecs-cluster
```

As we did with Kubernetes, we are tagging our newest image with the latest commit hash. Then we are using the sed tool to replace a placeholder with the string LAST_COMMIT in our task definition template with the latest image and its tag. We output that new file to a temp file, register a new task definition with it, and then update the service. When we update the service, it is not necessary to pass the revision number for the family name. If we don't pass one, the service will use the latest active revision number, which will be the one we are deploying at that moment. If you see the desired count, you'll see that we are only deploying one replica. That's because currently we have a replica running in the cluster and it's using the port 80. We cannot deploy two more replicas since we only have two instances in this cluster and this type of load balancer (classic) doesn't support dynamic port mapping, which allows us to have several containers exposing the same port to a port in the load balancer. If you want to have more replicas running, you should add more instances to the cluster.

Now we have to modify our task definition and add the placeholder. Open ecs/task-definitions/webapp.json and modify the image name to

```
"image": "pacuna/webapp:LAST_COMMIT"
```

And that's it for the `push.sh` file. Let's go with `migrate.sh`. Add the following to the `ecs/deploy/migrate.sh`:

```
#!/bin/sh
set -x
aws ecs run-task --task-definition webapp --overrides file://ecs/task-
definitions/migrate-overrides.json --cluster ecs-cluster
```

Pretty simple. We are just running the same command we ran to migrate our database. Again, we can just skip the revision number and ECS will run a copy with the latest revision. This simple command will run our latest migrations if there are any.

Now you can test those scripts by committing your changes

```
$ git add -A
$ git commit -m 'add deploy scripts for ecs'
```

And executing the files.

```
$ ./ecs/deploy/push.sh


++ git rev-parse --short HEAD
+ LC=1cc4379
+ docker build -f Dockerfile -t pacuna/webapp:1cc4379 .
Sending build context to Docker daemon 24.47 MB
Step 1 : FROM phusion/passenger-ruby23:0.9.19
 ---> 6841e494987f
 (truncated)
...
+ docker push pacuna/webapp:1cc4379
The push refers to a repository [docker.io/pacuna/webapp]
692dc46ce0e9: Pushed
b3e13bc936af: Pushing [======================>           ] 10.46 MB/23.17 MB
25c9e91cead0: Pushed
7570593f934a: Pushing [====================>             ] 9.678 MB/23.17 MB
(truncated)
...
{
    "service": {
        "status": "ACTIVE",
        "taskDefinition": "arn:aws:ecs:us-east-1:586421825777:task-
        definition/webapp:8",
        "pendingCount": 0,
        "loadBalancers": [
            {
                "containerName": "webapp",
                "containerPort": 80,
```

```
                "loadBalancerName": "webapp-load-balancer"
            }
        ],
...
```

And then migrate

```
$ ./ecs/deploy/migrate.sh

{
    "failures": [],
    "tasks": [
        {
            "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/a7128bec-
            43e9-4773-8425-698419d86db3\ ",
            "overrides": {
                "containerOverrides": [
                    {
                        "environment": [
                            {
                                "name": "PASSENGER_APP_ENV",
                                "value": "production"
                            }
                        ],
                        "command": [
                            "bin/rails",
                            "db:migrate",
                            "RAILS_ENV=production"
                        ],
                        "name": "webapp"
                    }
                ]
            },
            "lastStatus": "PENDING",
            "containerInstanceArn": "arn:aws:ecs:us-east-
            1:586421825777:container-instance/edd5e9a7-\ 7f29-43a6-98c1-
            b7d7dd912cd2",
            "createdAt": 1475198720.749,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/ecs-
            cluster",
            "desiredStatus": "RUNNING",
            "taskDefinitionArn": "arn:aws:ecs:us-east-1:586421825777:task-
            definition/webapp:8",
            "containers": [
                {
                    "containerArn": "arn:aws:ecs:us-east-
                    1:586421825777:container/bf8c369b-666a-42c6\
                    -8a3b-b57cca400c1b",
```

```
                    "taskArn": "arn:aws:ecs:us-east-1:586421825777:task/
                    a7128bec-43e9-4773-8425-6984\19d86db3",
                    "lastStatus": "PENDING",
                    "name": "webapp"
                }
            ]
        }
    ]
}
```

And that's it! Later we are going to use these scripts for building a pipeline with Jenkins. Jenkins will execute the scripts after every deploy we make to a GitHub branch.

Summary

After this chapter, you should be able to run your Rails application using Amazon ECS. You should also be able to see the main differences between ECS and Kubernetes and choose the solution that makes more sense to you. You can see that both technologies are pretty strong and have mostly conceptual differences. Although Kubernetes seems to have a better integration in some areas like Volumes and DNS, ECS is tightly integrated with all the AWS components, which makes it a very solid choice if you run your system on AWS. Just as we did with Chapter 3, we ended this chapter by creating a couple of automation scripts to run deploys in a very efficient and structured way.

In Chapter 5, we'll take automation to the next level by using a continuous integration server to run our deployments along with our test suite.

CHAPTER 5

Continuous Integration

In this chapter I'll show how you can automate the entire deployment process using Jenkins for both platforms, Kubernetes and ECS. Our final goal is to be able to update our application by pushing to some Git branch.

A typical continuous integration (CI) workflow when working with containers can be summarized as follows:

- A change is pushed to the application repository on GitHub.
- A service hook tells the CI server that a new version is available.
- The CI server pulls the latest changes from GitHub.
- The CI server builds a new Docker image with this new version.
- The CI server pushes that image to your DockerHub account.
- The CI server updates the necessary deployment files to use this new image version.
- The CI server applies the changes by making calls to the framework's API (application programming interface).

Those steps are basically what we have been doing manually until now, so it shouldn't be so hard to automate them. However, a big difference is that our current machine is completely configured to interact with all those different services. So the main difficulty with this pipeline will be to configure the CI server so it can connect to all the different services.

We'll use Jenkins for our CI server, mainly because it is highly configurable and it has very good support to work with tools like Docker and GitHub.

For Kubernetes, we'll have to install the **kubectl** tool and configure the proper cluster credentials, so it can interact with our cluster. Since we already have a couple of scripts that automate our main tasks, it'll be easy to build a new project to accomplish this.

For ECS we'll only need to configure the AWS CLI (Command Line Interface) with our credentials, the same way we did on our local machine. We also configured scripts that automate our tasks, so this is going to be very straightforward.

Let's start by installing Jenkins on a new machine.

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1_5

Installing Jenkins

For Jenkins, we are going to run a new AWS EC2 instance. We'll be using the AWS CLI for creating all the different resources we need. You can use the AWS Dashboard, but I prefer this approach since you can document everything and reuse the commands afterward. It's important to understand that Jenkins will be running on a regular EC2 Instance and not in a container. That's because Jenkins will have to build Docker images, and doing that from another container can be kind of tricky. So let's stay old school with our CI server and launch a regular server.

Creating a Key Pair

First let's create a new key pair for accessing the server. We can create and save one by running the following:

```
$ aws ec2 create-key-pair --key-name Jenkins --query 'KeyMaterial' --output
text > Jenkins.pem
```

Then we need to add the permissions for that `.pem`:

```
$ chmod 400 Jenkins.pem
```

Save this key in a safe folder. I like to keep my key pairs in `~/.ssh/keys`.

The instance we want to use has to be launched in a VPC (Virtual Private Cloud). We are going to use the same VPC where our cluster lives. If you don't remember how to get the security group and a subnet ID for this VPC, you can go back to the section where we created a database for our application running with ECS. You'll need your security group ID and one subnet ID.

Launching the Instance

You can use whatever image you prefer, but keep in mind that the Jenkins installation and general configuration may be different. I'll go with the `Amazon Linux AMI 2016.09.0 (HVM)` whose ID is `ami-c481fad3`. Let's use the following command for launching an instance with decent specs for a small Jenkins server in our existent VPC:

```
$ aws ec2 run-instances --image-id ami-c481fad3 --subnet-id subnet-3a09f717
--count 1 --instance-type t2.medium --key-name Jenkins --security-group-ids
sg-bbe6b3c1 --block-device-mappings '[{ "DeviceName": "/dev/xvda", "Ebs":
{ "VolumeSize": 20 } }]' --associate-public-ip-address
```

You'll have a big output showing you all new instance information:

```
{
    "OwnerId": "586421825777",
    "ReservationId": "r-445d77fa",
    "Groups": [],
```

```
"Instances": [
    {
        "Monitoring": {
            "State": "disabled"
        },
        "PublicDnsName": "",
        "RootDeviceType": "ebs",
        "State": {
            "Code": 0,
            "Name": "pending"
        },
        "EbsOptimized": false,
        "LaunchTime": "2016-09-29T19:16:17.000Z",
        "PrivateIpAddress": "10.0.1.116",
        "ProductCodes": [],
        "VpcId": "vpc-a0e9c0c7",
        "StateTransitionReason": "",
        "InstanceId": "i-729ea343",
        "ImageId": "ami-c481fad3",
        "PrivateDnsName": "ip-10-0-1-116.ec2.internal",
        "KeyName": "Jenkins",
        "SecurityGroups": [
            {
                "GroupName": "amazon-ecs-cli-setup-ecs-cluster-
                EcsSecurityGroup-1QI8JT422T2EQ",
                "GroupId": "sg-bbe6b3c1"
            }
        ],
        "ClientToken": "",
        "SubnetId": "subnet-3a09f717",
        "InstanceType": "t2.medium",
        "NetworkInterfaces": [
            {
                "Status": "in-use",
                "MacAddress": "12:d7:fb:9b:28:3b",
                "SourceDestCheck": true,
                "VpcId": "vpc-a0e9c0c7",
                "Description": "",
                "NetworkInterfaceId": "eni-bee16eac",
                "PrivateIpAddresses": [
                    {
                        "Primary": true,
                        "PrivateIpAddress": "10.0.1.116"
                    }
                ],
                "Attachment": {
                    "Status": "attaching",
                    "DeviceIndex": 0,
```

```
                        "DeleteOnTermination": true,
                        "AttachmentId": "eni-attach-111a42ba", "AttachTime":
                        "2016-09-29T19:16:17.000Z"
                    },
                    "Groups": [
                        {
                            "GroupName": "amazon-ecs-cli-setup-ecs-cluster-
                            EcsSecurityGroup-1QI8JT42\2T2EQ",
                            "GroupId": "sg-bbe6b3c1"
                        }
                    ],
                    "SubnetId": "subnet-3a09f717",
                    "OwnerId": "586421825777",
                    "PrivateIpAddress": "10.0.1.116"
                }
            ],
            "SourceDestCheck": true,
            "Placement": {
                "Tenancy": "default",
                "GroupName": "",
                "AvailabilityZone": "us-east-1c"
            },
            "Hypervisor": "xen",
            "BlockDeviceMappings": [],
            "Architecture": "x86_64",
            "StateReason": {
                "Message": "pending",
                "Code": "pending"
            },
            "RootDeviceName": "/dev/xvda",
            "VirtualizationType": "hvm",
            "AmiLaunchIndex": 0
        }
    ]
}
```

Now wait for a few minutes so the instance gets initialized. If you want, you can visit the AWS Console in order to check the instance status (or you can query the API if you prefer).

We can query the status using the instance ID from the previous output with

```
$ aws ec2 describe-instances --instance-ids i-729ea343
--query="Reservations[0].Instances[0].State"
```

Output:

```
{
    "Code": 16,
    "Name": "running"
}
```

Connecting to the Instance

First we'll need the public DNS (Domain Name System) or IP (Internet Protocol) of this new server. Run

```
$ aws ec2 describe-instances --instance-ids i-729ea343
--query="Reservations[0].Instances[0].NetworkInterfaces[0].Association.
PublicIp"
```

Output:

```
"184.72.110.253"
```

Now let's connect to that IP address using `ec2-user` as the user, and passing the Jenkins `.pem` created previously:

```
$ ssh ec2-user@184.72.110.253 -i Jenkins.pem
```

Make sure you replace the IP with yours and that the path for the key is correct. You should see the following welcome message:

```
 __|  __|_  )
 _|  (     /    Amazon Linux AMI
___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2016.09-release-notes/
No packages needed for security; 3 packages available
Run "sudo yum update" to apply all updates.
```

Let's install Jenkins and the other dependencies we need.

Installing Dependencies

Now that we are inside our server, let's install some tools we'll need. For that we'll log in as the root user and use `yum` to install these dependencies.

```
# sudo su
# yum update -y
# yum install -y git nginx docker
```

Let's also install Docker Compose with the instructions given in the releases page (https://github.com/docker/compose/releases).

```
# curl -L https://github.com/docker/compose/releases/download/1.8.1/docker-
compose-`uname -s`-`uname -m` > /usr/bin/docker-compose
# chmod +x /usr/bin/docker-compose
```

We'll be using Docker Compose to run our test suite later. Now let's add the Jenkins repository and install it.

```
# wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/
jenkins.repo
# rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
# yum install -y jenkins
```

Before starting the services let's add the Jenkins user to the Docker group.

```
# usermod -a -G docker jenkins
```

Now we can start the Jenkins and Docker services and add them to the system startup and run some commands so we can log in as the Jenkins user in case we need to.

```
# service jenkins start
# service docker start
# chkconfig jenkins on
# chkconfig docker on
# usermod -s /bin/bash jenkins
# usermod -m /var/lib/jenkins jenkins
```

The last dependency we need is **kubectl**. Without **kubectl** we can't talk to our cluster from Jenkins. We can install it with the following:

```
# curl -Lo kubectl http://storage.googleapis.com/kubernetes-release
/release/v1.3.0/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv
kubectl /usr/bin/
```

That's going to do the trick for Kubernetes. For ECS, we'll also need the AWS CLI. Let's install it with the following:

```
# curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-
bundle.zip"
# unzip awscli-bundle.zip
# ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
```

Before we restart the server, let's configure the main tools we need, Docker, **kubectl** for Kubernetes, and **aws cli** for ECS.

For Docker we only need to log in so this server can have access to DockerHub. We have to use the same credentials that allow access to the repository we have been working with.

```
# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
```

```
Username: pacuna
Password:
Login Succeeded
```

For **kubectl** we only need to add our cluster information and credentials so the Jenkins user can have access. All that information is kept in the `~/.kube/config` file of the user that's using the client. Let's switch to the Jenkins user and create that file.

```
# sudo su - jenkins
# mkdir -p ~/.kube
# touch ~/.kube/config
```

Now we can copy our local `~/.kube/config` file into the Jenkins server file. Remember that our local machine is already configured to interact with the cluster, so all of the necessary credentials live in that file. You should be careful with that information, since anyone who gets the file would have access to create and delete resources in your Kubernetes cluster. After you copy the file, you can remove the sections that belong to Minikube. For this server we only need access to our production cluster.

Your `~/.kube/config` in your Jenkins server should look something like the following:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: VERYLONGSTRING
    server: https://52.32.34.173
  name: aws_kubernetes
contexts:
- context:
    cluster: aws_kubernetes
    user: aws_kubernetes
  name: aws_kubernetes
current-context: aws_kubernetes
kind: Config
preferences: {}
users:
- name: aws_kubernetes
  user:
    client-certificate-data: VERYLONGSTRING
    client-key-data: VERYLONGSTRING
    token: 6YkrOgBXQXCgeATdTiAzc6diZk6VwMfR
- name: aws_kubernetes-basic-auth
  user:
    password: XXXXXXXXXXXXXXXX
    username: admin
```

Let's see if **kubectl** is correctly configured by getting the nodes of the cluster.

```
# kubectl get nodes
NAME                                         STATUS   AGE
ip-172-20-0-64.us-west-2.compute.internal    Ready    2d
ip-172-20-0-65.us-west-2.compute.internal    Ready    2d
```

Great! Now we have to configure the AWS CLI. This is going to be as simple as it was with **kubectl**. If you remember, the configuration for this tool lives in the files `~/.aws/credentials` and `~/.aws/config`. We also have to copy our local files that contain the access tokens for our account and our preferences to files in the home of the Jenkins user. Let's create the folder and files in the server. Remember you have to be logged in as the Jenkins user.

```
# mkdir -p ~/.aws
# touch ~/.aws/credentials
# touch ~/.aws/config
```

Now copy your local files to those remote files in the server. Let's test the configuration by querying the API and getting the cluster information.

```
# aws ecs describe-clusters --cluster ecs-cluster
{
    "clusters": [
        {
            "status": "ACTIVE",
            "clusterName": "ecs-cluster",
            "registeredContainerInstancesCount": 2,
            "pendingTasksCount": 0,
            "runningTasksCount": 2,
            "activeServicesCount": 1,
            "clusterArn": "arn:aws:ecs:us-east-1:586421825777:cluster/
            ecs-cluster"
        }
    ],
    "failures": []
}
```

You can see that the AWS CLI is correctly configured. Now we can reboot the server and start to work with Jenkins.

```
# reboot 1
```

After the reboot, Jenkins should be running on port 8080, but currently our VPC security group doesn't allow access through that port. Let's add a new rule.

```
$ aws ec2 authorize-security-group-ingress --group-id sg-bbe6b3c1 --protocol
tcp --port 8080 --cidr 0.0.0.0/0
```

Now we can go to http://184.72.110.253:8080 and see the Jenkins installer (Figure 5-1).

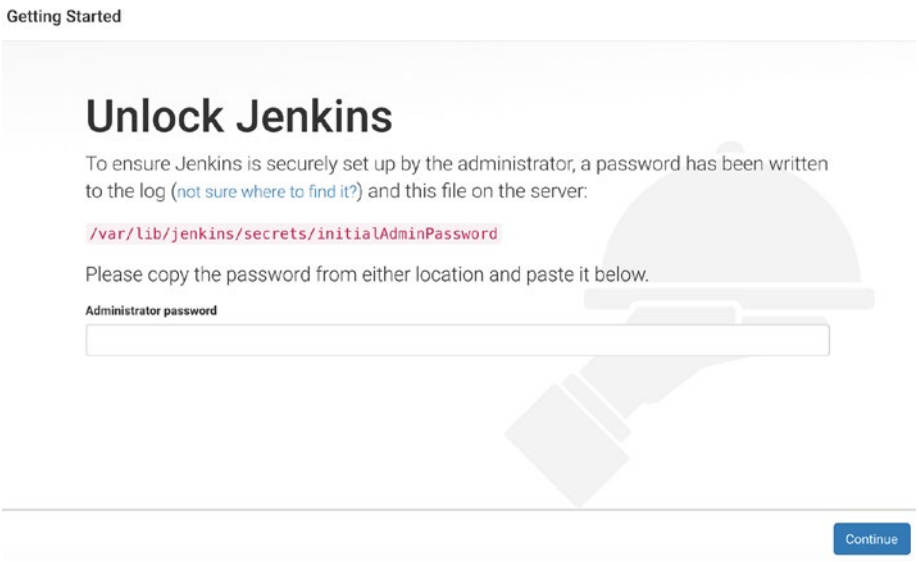

Figure 5-1. *Jenkins install*

Log in to the instance again and get that initial admin password.

```
$ ssh ec2-user@184.72.110.253 -i Jenkins.pem
# sudo su
# cat /var/lib/jenkins/secrets/initialAdminPassword
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Use that password to continue with the following steps. In the following step select **Install the suggested plug-ins**. Next, choose a username for the admin user along with a password and the other requested information. Then click **Save**, **Finish**, and finally **Start using Jenkins**.

Then you should see the Jenkins home page (Figure 5-2).

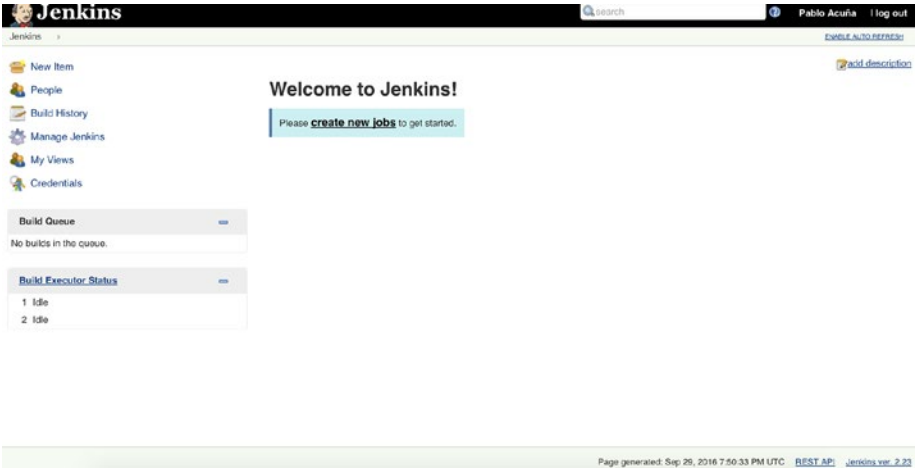

Figure 5-2. *Jenkins home page*

Now that we have our Jenkins server and our admin user configured, let's install a plug-in that will help us with the interaction with our GitHub repository.

Go to **Manage Jenkins ➤ Manage Plugins ➤ Available** and search for the following plug-in (see Figure 5-3):

```
GitHub Authentication plugin
```

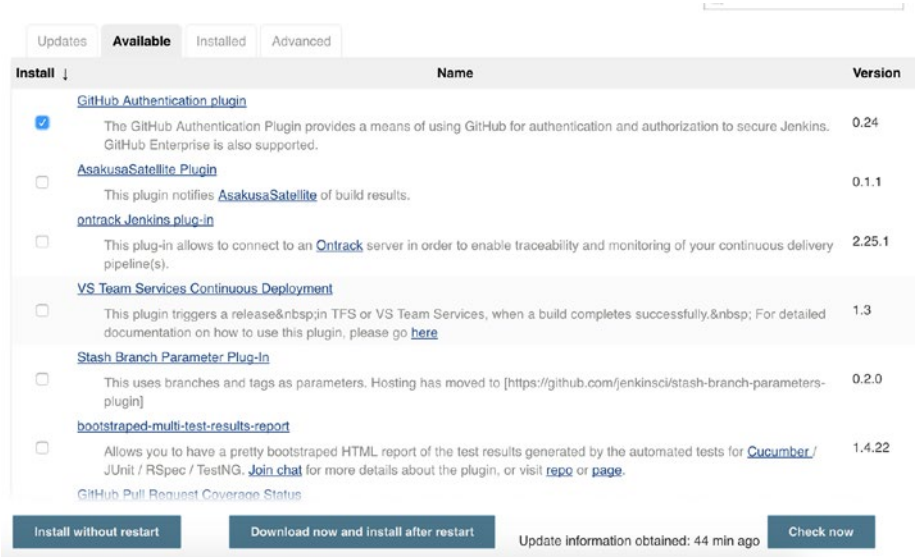

Figure 5-3. *Jenkins plug-ins*

Select the plug-in and click **Install without restart**.

Configuring a Job for Kubernetes

Right now we have our Docker Image, but we'll also need a GitHub repository so we can trigger a build after every deploy. Go to GitHub and create a new repository and call it `webapp`. In my case the URL (uniform resource locator) for the repo will be `https://github.com/pacuna/webapp`.

Go to the Jenkins home page (see Figure 5-2) and click **create new jobs**. Use the name `webapp-k8s`, select **Freestyle project**, and then click **OK** (see Figure 5-4).

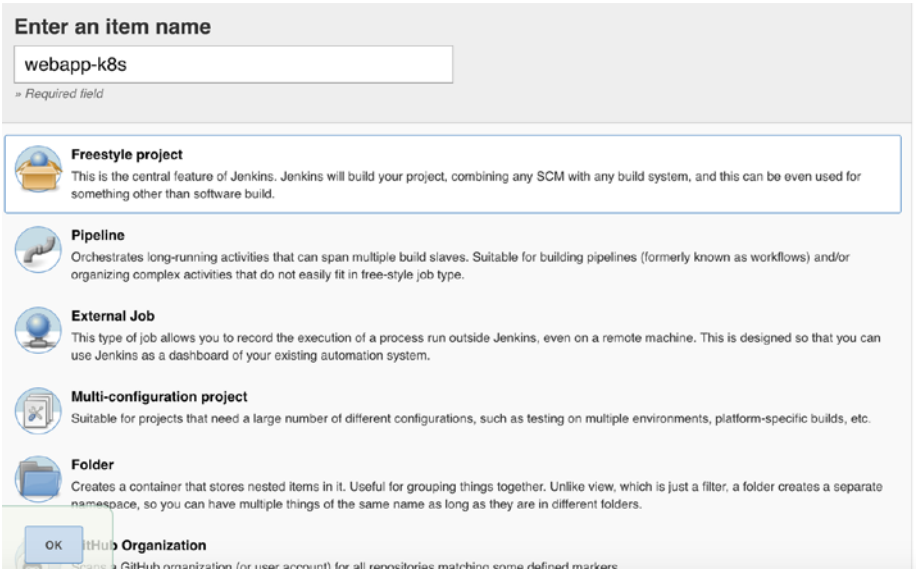

Figure 5-4. *Jenkins plug-ins*

In the **General** section (see Figure 5-5), check GitHub project and add the URL of your repository. For example, I'll use `https://github.com/pacuna/webapp`.

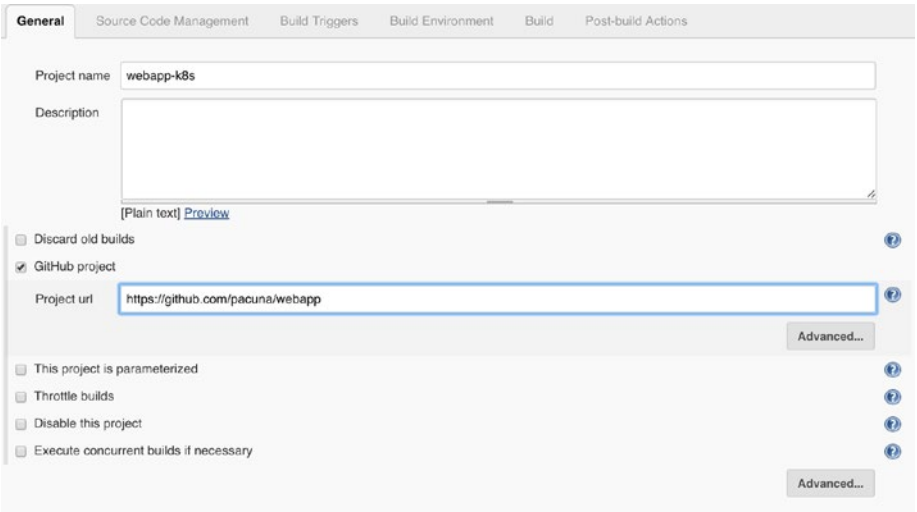

Figure 5-5. *Job configuration*

In the **Source Code Management** section (Figure 5-6) select **Git** and once again use your repository URL. Click the Add button in the credentials section to open the Jenkins credentials provider, use the **Username with password** kind, and add your username and password for GitHub. Make sure to select those credentials after, and that Jenkins doesn't throw any field errors.

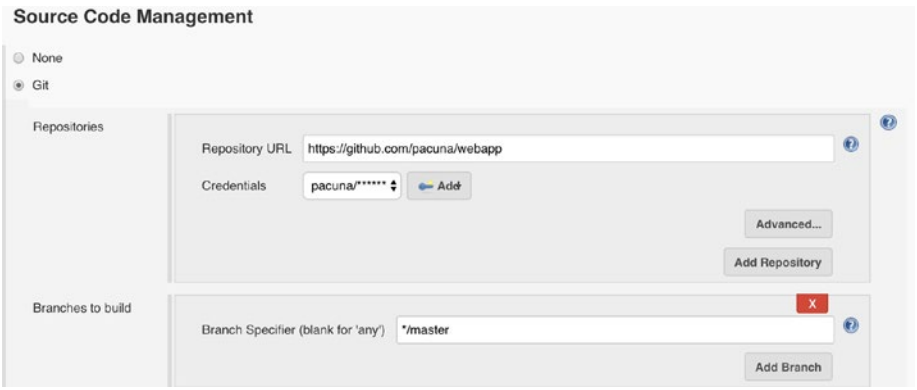

Figure 5-6. *Job configuration*

If you have problems with the GitHub authentication, you can also add your username and ssh private key to the Jenkins Credentials Provider and use the SSH URL of your repository.

In the **Build Triggers** section (see Figure 5-7), check **Build when a change is pushed to GitHub**.

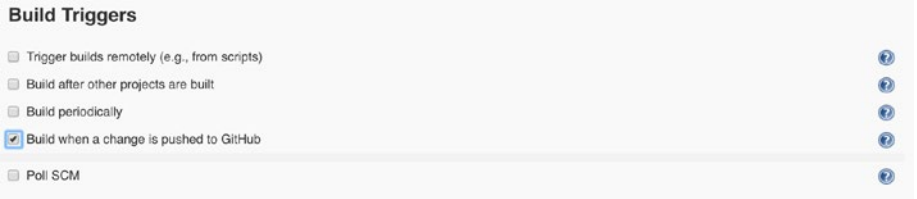

Figure 5-7. *Job configuration*

Finally, we get to the smart part of the pipeline. Click **Add build step** and select **Execute shell** (see Figure 5-8). This command will be executed after the latest code gets pulled from the GitHub repository. This script should be super simple. All the work is done by the two scripts we already have: `deploy/push.sh` and `deploy/migrate.sh`. So let's add a call to those two files:

Figure 5-8. *Job configuration*

Click **Apply** and then **Save**.

Our deployment pipeline for Kubernetes should be ready. Let's fix our database host in case you changed it while testing ECS.

```
production:
  <<: *default
  # host: webapp-postgres.caxygd3nhObk.us-east-1.rds.amazonaws.com
  host: postgres
  database: webapp_production
  username: webapp
  password: mysecretpassword
```

Let's commit that change.

```
$ git add .
$ git commit -m 'fix db host for kubernetes'
$ git push origin master
```

Remember we haven't added the hook to our GitHub repo. So right now a push to master will not trigger a deploy in Jenkins. We have to trigger a build by hand using the **Build Now** (see Figure 5-9) link on the sidebar of the project page in Jenkins. Click **Build Now** and you will see the job starting.

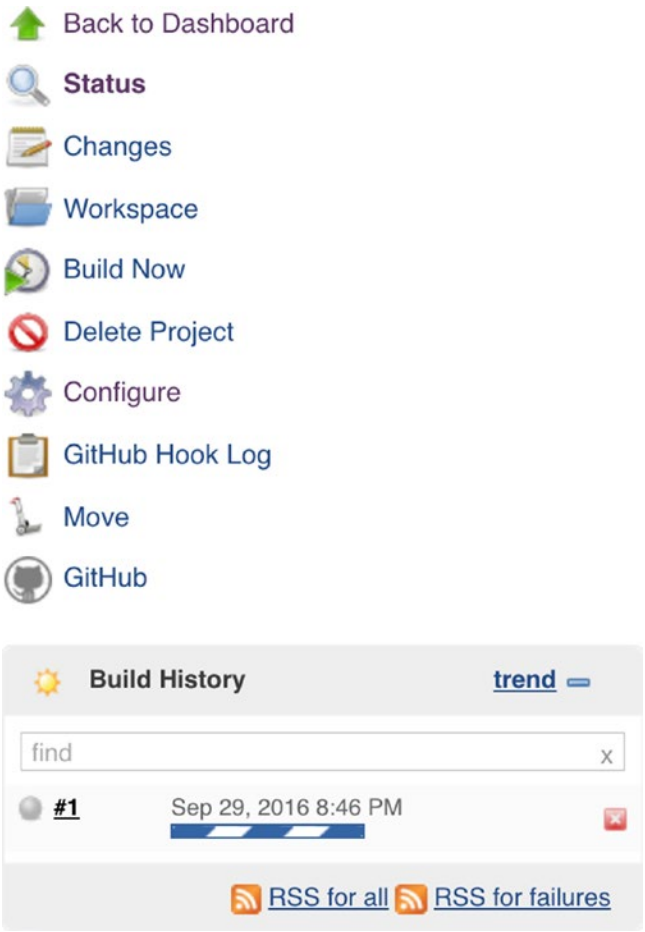

Figure 5-9. *Job configuration*

Click the job and then go to **Console Output** (Figure 5-10) to inspect the logs. There you should see all the output we used to inspect from our local machine but this time running on the Jenkins server.

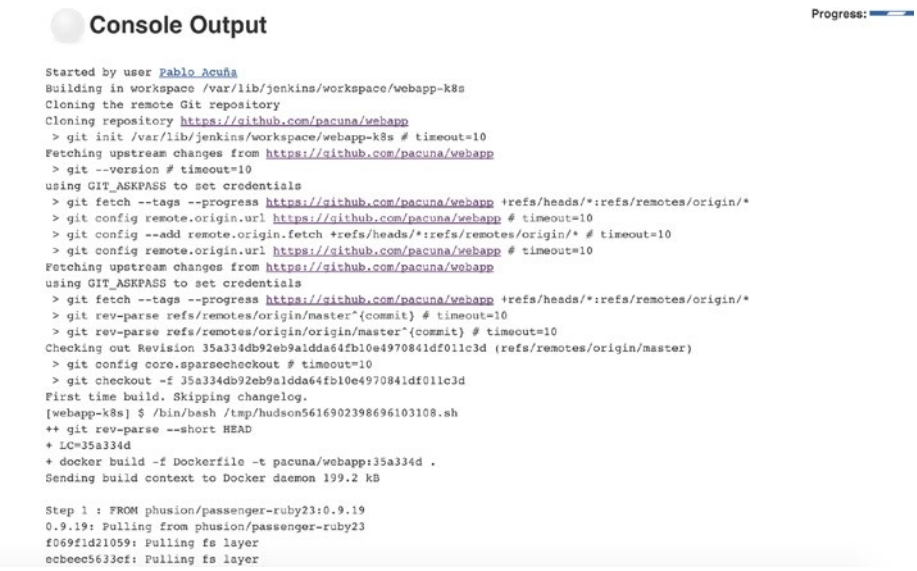

Figure 5-10. *Job configuration*

When the job finishes, you can go to the bottom of the logs and see **kubectl** updating our cluster (Figure 5-11).

```
56b80ccb87d2: Layer already exists
18c6e988b612: Layer already exists
9cf2c8cdd911: Layer already exists
61478a88b987: Layer already exists
206acb7055b4: Layer already exists
b9a186e5523b: Layer already exists
886837209eb3: Layer already exists
fb69f8349ff1: Layer already exists
e70ce51784fd: Layer already exists
28ba3922517b: Layer already exists
3b7983eb7687: Layer already exists
5f70bf18a086: Layer already exists
19a8383d6948: Layer already exists
872e268735cb: Layer already exists
0184e31d4eba: Layer already exists
0738910e0455: Layer already exists
21df36b5c775: Layer already exists
7f4734de8e3d: Layer already exists
315fe8388056: Layer already exists
35a334d: digest: sha256:1181be127531ff974563004a235ea91e02bc051495fedd7f38f2d55f1ed8bb3c size: 5094
+ kubectl set image deployment webapp webapp=pacuna/webapp:35a334d
deployment "webapp" image updated
++ git rev-parse --short HEAD
+ LC=35a334d
+ kubectl delete jobs/setup
job "setup" deleted
+ sed s/webapp:LAST_COMMIT/webapp:35a334d/g kube/jobs/setup-job.yaml
+ kubectl create -f setup-job.yaml.tmp
job "setup" created
+ rm setup-job.yaml.tmp
Finished: SUCCESS
```

Figure 5-11. *Job configuration*

Let's check if everything is OK with our webapp service. Remember you can get the load balancer DSN with `kubectl describe service webapp`. If you hit your service end point for articles you should get the correspondent response:

```
[{"id":1,"title":"my first article","body":"Lorem ipsum dolor sit
amet, consectetur adipiscing elit.\..","created_at":"2016-09-
27T03:07:36.706Z","updated_at":"2016-09-27T03:07:36.706Z","slug":null},{"i\
d":34,"title":"my second article","body":"Lorem ipsum dolor sit
amet, consectetur adipiscing elit...\ ","created_at":"2016-09-
28T03:20:37.427Z","updated_at":"2016-09-28T03:20:37.427Z","slug":"my-
second-\ article"}]%
```

Cool! The only missing thing is to add a Webhook to our GitHub repository, so a push to master triggers a deploy.

Push to Deploy

Go to your GitHub repository, click **Settings ➤ Integration & Services** and then **Add service**. Look for the Jenkins (GitHub plug-in) and select. Then in the **Jenkins hook ur**l add the following (replace with your Jenkins IP address) (see Figure 5-12):
`http://184.72.110.253:8080/github-webhook/`

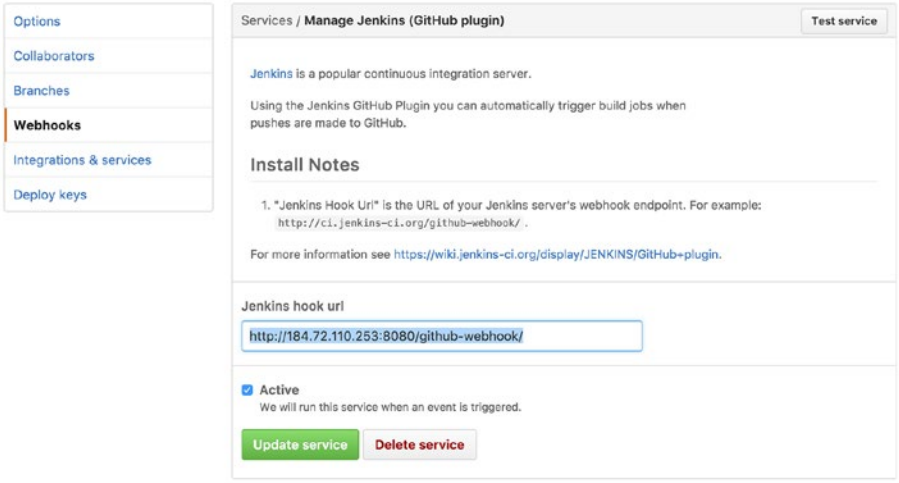

Figure 5-12. *Hook configuration*

Make sure you select **Active** and then click Save.

And that's it! Now every time you push something to your master branch, a deploy will be triggered for the associated Job we just configure.

Running the Test Suite

A big part in CI is testing. With Jenkins we are able to run the tests before updating our application.

We'll use Docker Compose to run our tests before we push the image to DockerHub. It doesn't make much sense to push a broken image. So this workflow will be the following:

- We push bad code to GitHub and trigger a new deploy.
- Jenkins pulls these changes.
- Jenkins runs Docker Compose using this new code and a custom entry point.
- Jenkins waits for the status code of the test container.
- If status is different than 0, the deployment stops.

First, let's try to run our tests locally. We'll have to create our test database in case it doesn't exist and also run the migrations. On your local machine run

```
$ docker-compose run --rm webapp bin/rails db:create RAILS_ENV=test
$ docker-compose run --rm webapp bin/rails db:migrate RAILS_ENV=test
```

And now let's run the following tests:

```
$ docker-compose run --rm webapp bin/rake RAILS_ENV=test
```

Output:

```
.

Finished in 1.267605s, 3.9444 runs/s, 5.5222 assertions/s.

5 runs, 7 assertions, 0 failures, 0 errors, 0 skips
```

Great! We have our tests passing locally. This was just to be sure that we have a test suite ready.

Let's create a Docker Compose file for the testing environment and a script for running a custom entry point (you can create these files in your root application folder).

```
$ touch docker-compose.test.yml
$ touch setup.test.sh
$ chmod +x setup.test.sh
```

First, for the `docker-compose.test.yml` file add the following:

```
version: '2'
services:
  webapp_test:
    container_name: webapp_test
    build: .
    depends_on:
      - postgres
    environment:
      - PASSENGER_APP_ENV=test
    entrypoint: ./setup.test.sh
  postgres:
    image: postgres:9.5.3
    environment:
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_USER=webapp
      - POSTGRES_DB=webapp_test
```

Pretty simple. We are running a container that builds our application from the latest code and is connected to a database container. We use the `setup.test.sh` as the new entry point, and we set the `PASSENGER_APP_ENV` to `test` so the commands in the entry point run on that environment.

Now, for the `setup.test.sh` file add the following:

```
#!/bin/sh

echo "Waiting PostgreSQL to start on 5432..."

while ! nc -z postgres 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

bin/rails db:create RAILS_ENV=test
bin/rails db:migrate RAILS_ENV=test
bin/rake RAILS_ENV=test
```

Just as with the setup container, we wait for the PostgreSQL container to be available. Then we create and migrate the test database and finally we run our tests with the `rake` command.

Now, just for testing purposes, let's break one of our tests.

Open the `test/controllers/articles_controller_test.rb` file and modify the status code of this test from 201 to 301.

```
test "should create article" do
  assert_difference('Article.count') do
    post articles_url, params: { article: { body: @article.body, title: @
    article.title } }, as: :json
  end

  assert_response 301
end
```

We just want to test that Jenkins will actually stop the deployment when it sees that this test fails. Now push this code to your GitHub repository. There's no need to build a new image since Jenkins is now doing that work for us with every build.

```
$ git add -A
$ git commit -m 'Add testing stuff'
$ git push origin master
```

Now let's go to our `webapp-k8s` project's configuration in Jenkins.

Add a new Build step selecting **Execute Shell** and then drag that step and put it before the one created previously. Remember we don't want to create and push the new image until we're sure the tests pass.

Add the following to that step:

```
#!/bin/sh

# create test environment
docker-compose -f docker-compose.test.yml build
docker-compose -f docker-compose.test.yml run --rm webapp_test

# check the last status code
if [ $? -eq 0 ]
then

        echo "All tests passed! :)"
else
        echo "Tests failed! :("
    exit 1
fi
```

Figure 5-13. *Test build step*

This script will run Docker Compose and will use the `docker-compose.test.yml` file we created. It'll build the image using the current workspace, which contains the latest code pulled from GitHub. If the tests failed, the exit code will be different than 0. In that case we stop the deployment using `exit 1` and Jenkins won't do anything else and will mark this build as failure. On the contrary, if the tests passes, the status code will be 0 and the deployment continues.

Click Apply and then Save. Now let's test this pipeline by running `Build Now` on the sidebar. If you go to the console output after build the project, at the end you should see

```
Failure:
ArticlesControllerTest#test_should_create_article [/home/app/webapp/test/
controllers/articles_controller_test.rb:18]:
Expected response to be a <301: Moved Permanently>, but was a <201:
Created>. Expected: 301
  Actual: 201

bin/rails test test/controllers/articles_controller_test.rb:13
...
(truncated)
...
.
```

```
Finished in 0.411073s, 12.1633 runs/s, 17.0286 assertions/s.

5 runs, 7 assertions, 1 failures, 0 errors, 0 skips
Tests failed! :(
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

That's excellent! Remember we pushed a broken test to GitHub and that's why this deploy is failing. Now let's fix the test, push the code to GitHub, and see if this build passes. First fix the articles controller test so it checks for a 201 status code.

```
test "should create article" do
  assert_difference('Article.count') do
    post articles_url, params: { article: { body: @article.body, title: @
article.title } }, as: :json
  end

  assert_response 201
end
```

Now push those changes to the repository.

```
$ git add .
$ git commit -f 'Fix broken test'
$ git push origin master
```

Now, go to your Jenkins projects to monitor the output of this build. At one point you should see your tests passing.

```
.

Finished in 0.416967s, 11.9914 runs/s, 16.7879 assertions/s.

5 runs, 7 assertions, 0 failures, 0 errors, 0 skips
All tests passed! :)

Then the image is built and pushed to DockeHub:

...
 docker build -f Dockerfile -t pacuna/webapp:9c387bc
Sending build context to Docker daemon   236 kB

Step 1 : FROM phusion/passenger-ruby23:0.9.19
 ---> 6841e494987f
Step 2 : ENV HOME /root
 ---> Using cache
 ---> 36d6c73cd6e5
```

```
Step 3 : CMD /sbin/my_init
 ---> Using cache
 ---> d94279fdbbc8
Step 4 : RUN apt-get update && apt-get install -y -o Dpkg::Options::=
"--force-confold" netcat
 ---> Using cache
 ---> e0d7432f73c6
Step 5 : RUN rm -f /etc/service/nginx/down
 ---> Using cache
 ---> b8a8f4e592e5
...
```

And finally the Kubernetes cluster is updated.

```
+ kubectl delete jobs/setup
job "setup" deleted
+ sed s/webapp:LAST_COMMIT/webapp:9c387bc/g kube/jobs/setup-job.yaml
+ kubectl create -f setup-job.yaml.tmp
job "setup" created
+ rm setup-job.yaml.tmp
```

Awesome! Now you know how to add a build step that runs your test suite for avoiding deploys that may break your application.

You may want to version all of the scripts that you use in your Jenkins steps. It's better to keep those scripts in your applications and only call the file instead of writing the bash code in Jenkins. But that's up to you. I'm only giving you some ideas that can help you to write your customized scripts.

Configuring a Job for ECS

This configuration will also be pretty straightforward. The advantage we have is that we already have done all that we need. Similarly to what we did for the Kubernetes job, we are going to use the automation scripts we have and add the same step to run the test suite.

Now, chances are you are only going to use either Kubernetes or ECS, but not both. So while we build this ECS pipeline, we'll have to change code that's going to invalidate the Kubernetes pipeline. For example, the database host is not the same. For Kubernetes we were using an internal service with an alias of `postgres`, and for ECS we are using an RDS end point. Also, we cannot trigger two different hooks for our GitHub repo because we would be building a wrong image for one the platforms. With all of that said, let's begin.

Go to the Jenkins home and create another job. This time give it the name `webapp-ecs` and choose `Freestyle project`. If you have doubts with the graphical interface for Jenkins, take a look at the **Configuring a Job for Kubernetes** section. There you can see a couple of screenshots of the steps. Click **OK**.

For the general section, select **GitHub project** and add the project URL. In my case I'll use `https://github.com/pacuna/webapp/`.

In the **Source Code Management** section, select Git and add the same repository URL. If you haven't configured your GitHub credentials, click add and fill out with your GitHub username and password. Then select those credentials for your repo. If the credentials aren't working, you'll see an error.

In the **Build Triggers** section, choose **Build when a change is pushed to GitHub.**

Now skip to the **Build** section and add a build step of type **Execute Shell**. Add the following script to the text area:

```
#!/bin/bash
set -e

./ecs/deploy/push.sh && ./ecs/deploy/migrate.sh
```

As you can see, we are calling our two scripts that will update the cluster with the latest code. Jenkins will pull the latest changes from GitHub and using those scripts will build a new Docker image, push it to DockerHub, and then update the task definition and service in our cluster.

And that's it. Click **Apply** and then **Save**.

Now in your Project page, click **Build Now** in the sidebar. This will start a new build for the project. Click the new job that started and then click **Console Output**. You should see the entire deployment from the pull from GitHub to the update of the task definition and service, just as on our machine.

Running the Test Suite

Now let's add another build step that's going to run our tests. Go back to your project's configuration page and add a new build step of the same type, **Execute Shell**, and put it on before the other build step. We want to run the test suite before creating the new image. Add the following in the text area:

```
#!/bin/sh

# create test environment
docker-compose -f docker-compose.test.yml build
docker-compose -f docker-compose.test.yml run --rm webapp_test

# check the last status code
if [ $? -eq 0 ]
then

          echo "All tests passed! :)"
else
             echo "Tests failed! :("
      exit 1
fi
```

Our `docker-compose.test.yml` should be

```
version: '2'
services:
  webapp_test:
    container_name: webapp_test
    build: .
    depends_on:
      - postgres
    environment:
      - PASSENGER_APP_ENV=development
    entrypoint: ./setup.test.sh
  postgres:
    image: postgres:9.5.3
    environment:
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_USER=webapp
      - POSTGRES_DB=webapp_test
```

And the `setup.test.sh` should be

```
#!/bin/sh

echo "Waiting PostgreSQL to start on 5432..."

while ! nc -z postgres 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

bin/rails db:create RAILS_ENV=test
bin/rails db:migrate RAILS_ENV=test
bin/rake RAILS_ENV=test
```

If you haven't created the `docker-compose.test.yml` and the `setup.test.sh` files, go to the test suite part of the Kubernetes job's section. That script will run our test suite and check for the status code of the process. If the tests fail, the deploy will be interrupted. If the tests pass, the deployment will continue and the new image will be deployed in the cluster.

The final part is to create a hook for triggering build when pushing to the GitHub repository. For that, please take a look at the **Push to Deploy** section for the Kubernetes job. The process is the same.

Summary

In this chapter we saw how a CI server can helps us to create a deployment pipeline to run our containerized architecture. You saw the power and flexibility of Jenkins as a CI server and the little amount of configuration that it required. We created all the needed resources with the AWS CLI so you could have a fast mechanism to launch them; also we used several bash scripts in our build steps so our Jenkins jobs were as simple as possible. This also allows us to replicate and create new jobs very easily.

A very important aspect of CI is to correctly understand the workflow and the life cycle of a deployment. We started by pulling the latest changes from our GitHub repository; we then generated a new image which is pushed to our DockerHub repository; finally, we updated the corresponding templates to apply the changes in the containers within the cluster.

We used Docker Compose to add a new build step where we ran our test suite. This prevented pushes that break tests by stopping deployments that return errors from this step.

Index

P. Acuña, *Deploying Rails with Docker, Kubernetes and ECS*,
DOI 10.1007/978-1-4842-2415-1

G, H

I

J

K

L

M

N, O

P, Q

R

S

T

U

V

W, X

Y, Z