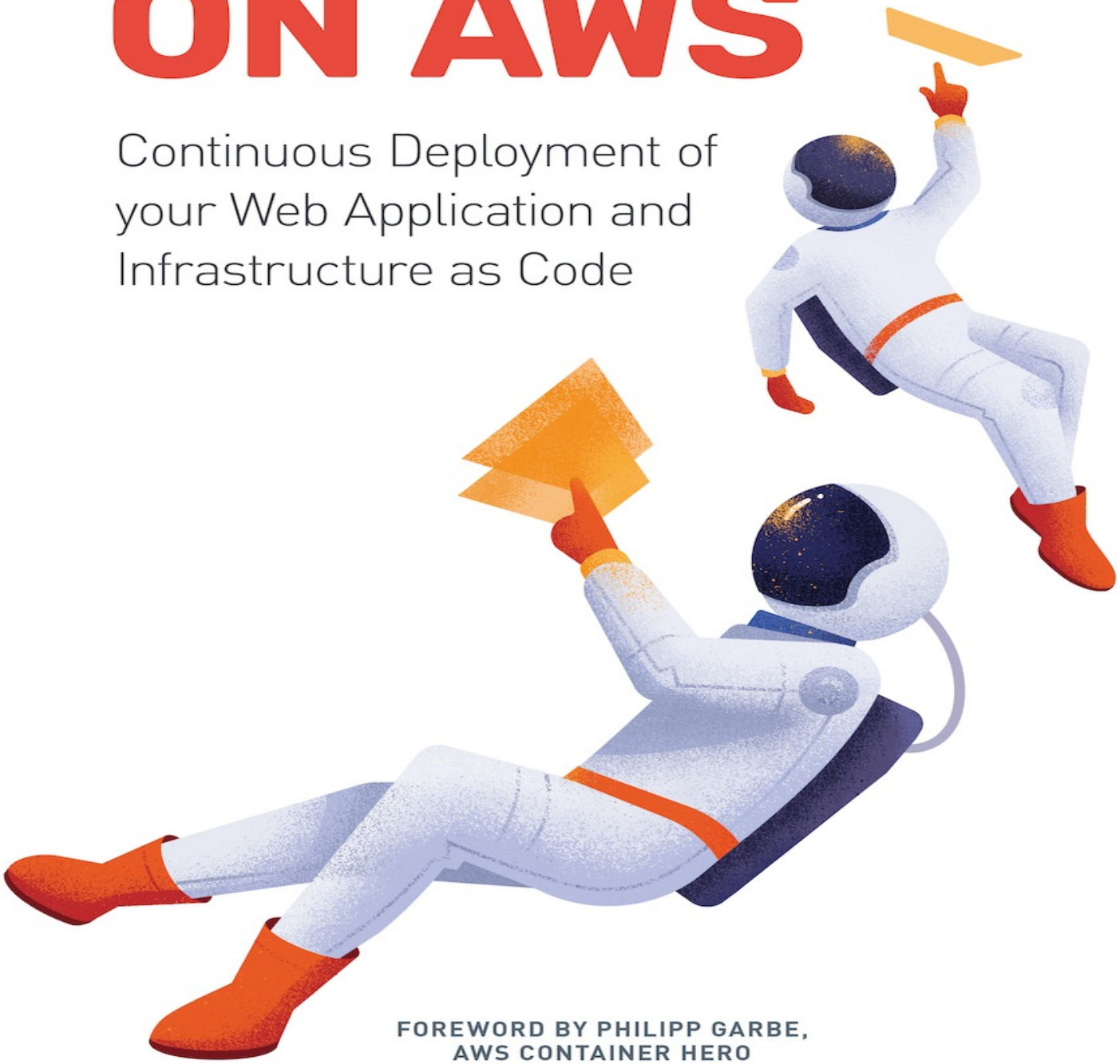
ANDREAS & MICHAEL WITTIG

RAPID DOCKER ON AWS

Continuous Deployment of your Web Application and Infrastructure as Code



FOREWORD BY PHILIPP GARBE, AWS CONTAINER HERO

Foreword

When Solomon Hykes presented Docker for the first time in 2013, its success could not yet be foreseen. However, developers immediately loved Docker's simplicity, and its success was unstoppable. Today, nearly every developer works with containers in some way or another.

It took some time until it was possible to deploy Docker containers efficiently in production. Things changed when the first orchestration tools became available. Docker built Swarm, Google open sourced Kubernetes, and AWS released the Elastic Container Service (ECS).

Nowadays, there are many options for deploying container workloads. This book does not blindly follow any technology hype, but gives you an opinionated blueprint for running container-based applications on AWS in a highly available, cost-effective and scalable manner with as little operational effort as possible. It covers not only Fargate but also other AWS services like Application Load Balancer (ALB), Relational Database Service (RDS), and CloudWatch to demonstrate what is needed for production readiness.

What I personally like very much is that the whole infrastructure is defined as code and comes with the book. It's also great to see that version control and continuous deployment are covered in their own chapter.

The two authors of the book, Andreas and Michael, have been working with AWS for many years and have gained a lot of experience. They are active in the community, sharing many of their experiences for free on their blog as well as in their production-ready CloudFormation templates. I had the opportunity to work with them for several months and much of what I know about AWS today I owe to them!

Now, get your hands dirty and deploy your containers rapidly on AWS!

Philipp Garbe, AWS Container Hero

About the book

Welcome Dzenan,

this is a rapid way to get your web application up and running on AWS. Made for web developers and DevOps engineers who want to dockerize their web applications and run their containers on Amazon Web Services. Prior knowledge of Docker and AWS is not required.

What you will learn while reading this book and working with the (code) examples:

- Running your web application on AWS with Docker: ECS and Fargate.
- Minimizing downtime with health checks and load balancing: ALB.
- Scaling your web application requires a scalable database: RDS Aurora Serverless.
- Minimizing operational effort with managed services.
- Reducing costs and minimizing paying for idle resources.
- Dockerizing your web application (PHP, Ruby, Python, Spring Boot, and Node.js).
- Monitoring and debugging with CloudWatch.
- Securing your web application with HTTPS.
- Leveraging infrastructure as code to automate your stack: AWS CloudFormation.
- Versioning your source code with Git: AWS CodeCommit.
- Deploying your source code and infrastructure continuously: AWS CodeBuild.

About us

We are two brothers and long-time AWS users, consultants, authors, and teachers. In this book, we share our experience with you. However, we don't want to bore you with technical details and dozens of alternatives. Instead, we present to you the best way to run Docker workloads on AWS including the source code that you need to get your own application up and running. *The best way* is opinionated and the essence of our experience.

Community and Feedback

Join the community of readers and us at https://community.cloudonaut.io: ask questions, share your learnings, and give feedback.

Conventions used in this book

The following boxes are designed to provide additional information or to get your attention.

> **Note** Additional information for the interested reader that is not required to understand the book.

> **Dependency** To read on, you must meet certain requirements. Important for readers who do not read linearly.

> **Warning** Read warnings carefully!

> **Customization** A box indicating that you might need to customize something here in your project.

Code, commands, and file names, are formatted in a `fixed-with font`. Bash commands that are too long to fit a line in the book are wrapped like this:

```
ls \
  -l
```

You can copy and paste them without modifications into your shell.

> **Note** The `\` backslash escapes the next character (a new line character here) from being interpreted by the shell.

Sometimes we shorten output or long strings with `[...]` to increase readability.

Thanks!

First of all, we want to thank Andrew Clark for editing. Andrew improved our teaching, our grammar, and style, as well as our examples enormously.

We also want to thank Philipp Garbe for writing such a benevolent foreword to our book.

Thanks to all the early birds, who bought this book long before we completed all chapters and examples.

Thanks to you for buying this book. We hope you will enjoy it as much as we do.

Thanks for spreading the word!

Andreas and Michael

Copyright, License and Trademark

widdix GmbH
Blumenstraße 17/1
73669 Lichtenwald
Germany

Many of the designations used by Amazon Web Services and other vendors to distinguish their products are claimed as trademarks. Where those designations appear in the book, and widdix GmbH was aware of a trademark claim, the designations have been printed in initial caps. The following are trademarks of Amazon.com, Inc. or its affiliates: Amazon Web Services, AWS, Amazon Athena, Amazon Aurora, Amazon CloudWatch, Amazon Cognito, Amazon DynamoDB, Amazon EC2, Amazon Elastic Compute Cloud, Amazon RDS, Amazon Relational Database, Amazon Route 53, Amazon S3, Amazon Simple Notification Service, Amazon Simple Storage Service, Amazon Virtual Private Cloud, Amazon VPC, AWS CloudFormation, AWS CodeBuild, AWS CodeCommit, AWS Step Functions, DynamoDB, and EC2.

Licensed to Dzenan Dzevlan (dzenan.dzevlan@gmail.com)

Introducing the highly available, scalable, and cost-effective architecture

In this chapter, you start 10,000 feet high looking down at the Rapid Docker on AWS architecture from a business and technical perspective. Afterward, you will set up your machine to work with the book's source code. At the end of the chapter, you will have a demo application up and running.

Benefits of the architecture

When designing the Rapid Docker on AWS architecture, we had one goal in mind: a production-ready infrastructure for everyone. We went through a long, iterative design process. We were inspired by numerous customer projects where customers asked us to minimize operational effort or shorten time to market. For years, we have been waiting for new AWS features that would help us achieve our ultimate goal. Now, we are happy to share our solution, which has the following benefits.

Low operational effort

All AWS building blocks used in the architecture described in this book are fully managed services causing minimal operational effort. You don't have to sign TLS certificates or install a database engine. You only have to care about your Docker image with your configuration, source code, and dependencies. AWS even takes care of your data by performing daily backups. AWS also covers security aspects, such as data encryption in-transit (using HTTPS/TLS) and at-rest. In a nutshell, you don't need highly specialized people on your team to operate the infrastructure.

Ready for the future

Architectures are not static anymore. An evolutionary architecture evolves with new requirements. When making use of infrastructure as code (IaC), you can deploy changes to your architecture with confidence. IaC enables you to test changes to your architecture before you apply them to production.

Cost-effective

In an ideal world, you only pay when your system is processing a user request. You never want to pay for idle resources. The architecture described in this book minimizes your costs for idle resources. The database can stop if not used, and you only run as few containers as possible to provide the needed performance.

Highly available

Sooner or later, a single point of failure causes an outage. This Rapid Docker on AWS architecture has no single point of failure and uses redundant components everywhere, from the load balancer to the multiple containers running on different hosts to the redundant database cluster.

Scalable

Your infrastructure is only as powerful as your weakest component. It is not sufficient to scale your web servers to handle a growing amount of requests. You have to scale your database as well. That's why the architecture described in this book scales the whole stack: the load balancer, the Docker containers running web servers, and the database.

Overview of the architecture

The main building blocks of the Rapid Docker on AWS architecture are:

- A load balancer: Application Load Balancer (ALB)
- Docker containers: Amazon ECS & AWS Fargate
- A database: Amazon Aurora Serverless (MySQL-compatible edition)

The following figure shows the high-level architecture:

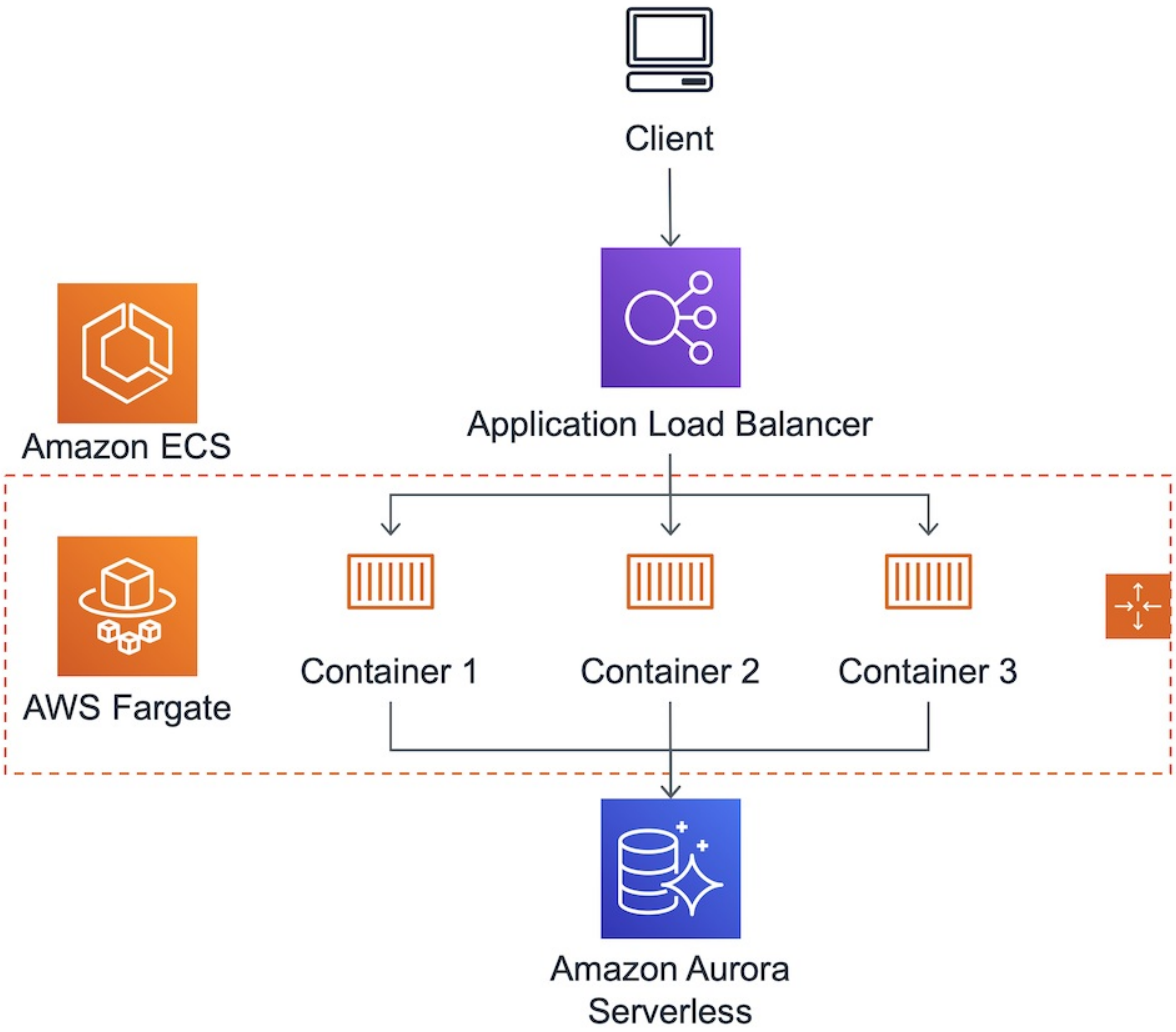


Figure 1: High-level architecture of the web application

You'll now get a short introduction into each of the building blocks. AWS prices

are given for the N. Virginia (us-east-1) region and vary across regions.

Application Load Balancer

The Application Load Balancer (ALB) distributes HTTP and HTTPS requests among a group of hosts. In our case, the hosts are Docker containers running a web server. An ALB is highly available and scalable by default, and it serves as the entry point into your infrastructure. The traffic flows from the client to your containers as follows:

1. The client makes an HTTP(S) request to the load balancer.
2. The load balancer receives the request and distributes it to one of the containers running a web server.
3. The containerized web server receives the request, does some processing, and sends a response.
4. The load balancer receives the response and forwards it to the client.
5. The client receives the response.

ALB pricing is based on two dimensions. First, you pay $0.0225 per hour (or partial hour) no matter how much traffic it serves. Second, you pay for the higher number of new connections, active connections, or traffic. Sound confusing? You can use the Simple Monthly Calculator to estimate your costs as shown in the following figure:

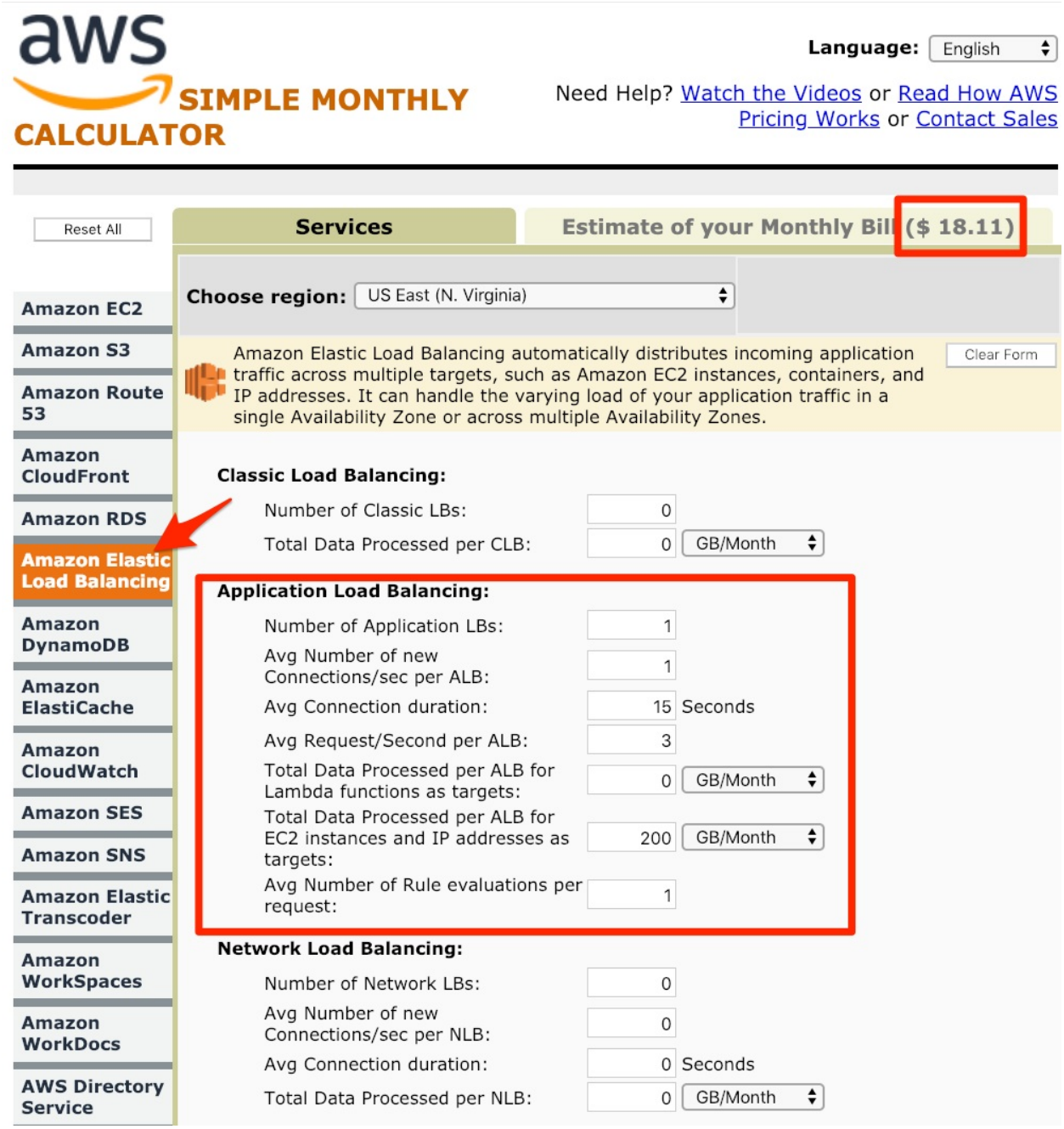


Figure 2: ALB cost estimation with the Simple Monthly Calculator

The important thing to know is that you always pay $16.20 per month (30 days) for your load balancer. Traffic is usually the second dimension that you have to pay for as well.

Continue on to learn about the Docker aspect of the architecture.

Amazon ECS & AWS Fargate

Amazon ECS is a fault-tolerant and scalable Docker container management service that is responsible for managing the lifecycle of a container. It supports two different modes or "launch types," one of which is AWS Fargate, which eliminates the need for managing a cluster of EC2 instances yourself. Instead, the cluster is managed by AWS, allowing you to focus on the containers rather than on the underlying infrastructure.

> **Note** Besides Fargate, you can also use EC2 instances to run Docker containers with ECS. We will not cover ECS with EC2 as it causes much more operational effort.

ECS is free of charge. Fargate pricing is based on the allocated vCPU and memory over time. A vCPU costs $0.04048 per hour and a GB of memory costs $0.004445 per hour. If you provision 0.25 vCPU and 0.5 GB memory for your container, it costs you $8.89 per month (30 days). To avoid a single point of failure, you should always run two containers at a time, totaling $17.78 per month. Pricing is per second with a minimum of 1 minute.

Unfortunately, the Simple Monthly Calculator does not support Fargate.

Amazon Aurora Serverless

Amazon Aurora Serverless is a highly available, scalable, MySQL 5.6-compatible relational database cluster. Depending on the load (CPU utilization and the number of connections), the cluster grows or shrinks within configurable boundaries. Aurora Serverless can scale down to zero when there are no connections for a configurable timespan.

Aurora Serverless pricing is based on three dimensions:

1. Compute and memory capacity measured in Aurora Capacity Units (ACUs). One ACU has approximately 2 GB of memory with an undocumented share of CPU and networking. You pay $0.06 per ACU per hour with a minimum of one ACUs per cluster. If the cluster is stopped you consume zero ACUs.
2. Storage: $0.10 per GB per month.
3. I/O rate: $0.20 per 1 million requests.

If your database is of modest size (e.g., 50 GB), you would pay $5 per month for

storage. On average, two ACUs should be sufficient for a modest database, which costs another $86.40 per month if your database runs 24/7. However, many databases are not needed 24/7 and can be stopped if not used like dev systems or internal systems that are used during working hours only.

You can use the Simple Monthly Calculator to estimate your exact costs.

Next, you prepare your machine.

Preparing your machine

Before you can launch the demo infrastructure and application, you have to prepare your machine.

> **Dependency** If you don't have an AWS account, create one at https://aws.amazon.com/.

The following steps are necessary:

1. Install Docker on your machine
2. Configure your temporary working environment
3. Create an S3 bucket for artifacts

Installing Docker

Open a terminal (on Windows, open PowerShell) to test if Docker is already installed. Run the following command:

```
docker -v
```

If you get an error like this:

```
command not found
```

Follow the official Docker installation guide tailored to your operating system to install the Community Edition (CE):

- Mac
- Windows:
  - Windows 10 (64bit, Pro, Enterprise or Education)
  - Others
- Linux:
  - CentOS
  - Debian
  - Fedora
  - Ubuntu
  - Others

Otherwise, Docker is already installed, and you will get an output like this:

```
Docker version 18.09.1, build 4c52b90
```

It's time to start your first Docker container to see if your Docker setup is complete. Run the following command:

```
docker run hello-world
```

If you get output like this:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:[...]4621577a99efb77324b0fe535
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
[...]
```

Docker is working correctly and you can go to the next section. If you get an error like this:

```
docker: Cannot connect to the Docker daemon at
unix:///var/run/docker.sock. Is the docker daemon running?
```

Docker is already installed on your machine but not started.

- Mac: Start the Docker application
- Windows:
  - Windows 10 (64bit, Pro, Enterprise or Education): Start the Docker Desktop for Windows application
  - Others: On your Desktop, click the Docker QuickStart icon to launch a pre-configured Docker Toolbox terminal
- Linux: In your terminal, run `sudo systemctl start docker` or `service docker start`

Wait for Docker to finish starting up and then retry the command:

```
docker run hello-world
```

Configuring your temporary working environment

This book comes with a Docker image that contains everything you'll need to run the examples in this book. Together with the book's source code, you can run all the examples.

> **Note** The Docker image is based on Amazon Linux 2 and contains all the tools you need in this book.

Follow these steps to set up your temporary working environment:

On Mac and Linux

1. Download the book's source code
2. Unzip the archive
3. Open a terminal
4. Change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment. When you run the command for the first time, you may need to allow Docker to share your drive.

```
bash start.sh
```

On Windows

1. Download the book's source code
2. Unzip the archive
3. Open PowerShell
4. Change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment. When you run the command for the first time, you may need to allow Docker to share your drive.

```
powershell -ExecutionPolicy ByPass -File start.ps1
```

Exploring your temporary working environment

You have now started a Docker container (aka your temporary working environment). You should see the following output:

```
bash-4.2#
```

Type `ls` into the terminal and hit enter. You should see folders and files from the book's source code.

To quit your temporary working environment type `exit` and hit enter. To spawn a new temporary working environment, re-run the start command that you used earlier.

You mounted your current working directory (containing the book's source code) into the container. Therefore, if you change a file in your Docker container, it also changes in your current working directory and vice versa. In addition to the source code, you also mounted a special `.aws` directory from your user's home directory. The `.aws` folder contains the configuration of the AWS CLI.

Run the following command in your temporary working environment to test your AWS CLI setup:

```
aws sts get-caller-identity
```

> **Warning** If you use more than one AWS account, check if the AWS account ID is correct. If not, you'll want to check out named profiles.

If the output looks like this you can jump to the next section.

```
{
  "Account": "123456123456",
  "UserId": "XXXXXXXXXXXXXXXXXXXXX",
  "Arn": "arn:aws:iam::123456123456:user/username"
}
```

If you get an error like this:

```
Unable to locate credentials. You can configure credentials by
running "aws configure".
```

You have to configure the AWS CLI. Follow the official Configuring the AWS CLI guide from AWS.

Once the AWS CLI setup is complete, continue with the next step.

Creating an S3 bucket for artifacts

To deploy your infrastructure, you'll need to upload artifacts to Amazon S3. Artifacts are deployable components of your application (e.g., a ZIP file of your PHP code, a JAR file). S3 is an AWS service to store files.

It's time to create your artifacts bucket. Please follow the naming convention to make it easier for you to work with the book. Replace the `$NICKNAME` variable with your nickname (only use characters [a-z0-9]).

> **Warning** All examples in this book are tested in `us-east-1` and `eu-west-1`. If you use a different region check if all services are supported in your region. You can query your default region with `aws configure get region` and change it with `aws configure set region us-east-1`.

The following command creates an S3 bucket:

```
aws s3 mb s3://docker-on-aws-$NICKNAME
```

A reader named Sally born 1987 could execute:

```
aws s3 mb s3://docker-on-aws-sally1987
```

If you get an output like this:

```
make_bucket: docker-on-aws-$NICKNAME
```

You're good to go. Move on to the next section. If you get an error like this:

```
make_bucket failed: s3://docker-on-aws-$NICKNAME An error
  occurred (BucketAlreadyExists) when calling the CreateBucket
  operation: The requested bucket name is not available. The bucket
  namespace is shared by all users of the system. Please select
  a different name and try again.
```

Choose a different `$NICKNAME` and try again.

You've completed the preparations! It's time for the real action.

Launching the demo infrastructure and application

In this section, you deploy the demo web application into your AWS account. You use the infrastructure as code (IaC) tool AWS CloudFormation to create all the resources. Don't be frightened by the fact that you don't know how CloudFormation works yet. Just enjoy the show.

> **Warning** Costs arise when you launch the demo infrastructure and application. You can expect costs of around $4.50 per day. You will learn to delete all AWS resources at the end of a section or chapter.

In your temporary working environment (replace `$NICKNAME`), execute the following commands:

```
cd /src/demo-app/aws/
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
aws cloudformation deploy --template-file .template.yml \
  --stack-name demo-app --capabilities CAPABILITY_IAM
```

The last command takes around 20 minutes to complete with an output like this:

```
Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - demo-app
```

The AWS resources are created. The following command fetches the URL:

```
aws cloudformation describe-stacks --stack-name demo-app \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

The output looks like this:

```
http://demo-LoadB-[...].elb.amazonaws.com
```

Open the URL in your web browser. The following figure shows a screenshot of the demo web application.

Figure 3: Demo web application screenshot

Remember all the benefits of the Rapid Docker on AWS architecture. Isn't it amazing how easy you can launch this architecture?

It's time to delete the running demo web application to avoid future costs. Execute the following commands (takes around 15 minutes to complete):

```
cd /src/demo-app/aws/
bash cleanup.sh
```

Next, replace the demo application with your own application.

Dockerizing and spinning up your own web application

Shipping software is a challenge. Endless installation instructions explain in detail how to install and configure an application as well as all its dependencies. But in practice, following installation instructions ends with frustration: the required version of PHP is not available to install from the repository, the configuration file is located in another directory, the installation instructions do not cover the operating system you need or want to use, etc.

And it gets worse: to be able to scale on demand and recover from failure on AWS we need to automate the installation and configuration of our application and its runtime environment. Implementing the required automation with the wrong tools is very time-consuming and error-prone.

But what if you could bundle your application with all its dependencies and run it on any machine: your MacBook, your Windows PC, your test server, your on-premises server, and your cloud infrastructure? That's what Docker is all about.

In short: Docker is a toolkit to deliver software.

Or as Jeff Nickoloff explains in Docker in Action (Manning): "Docker is a command-line program, a background daemon, and a set of remote services that take a logistical approach to solving common software problems and simplifying your experience installing, running, publishing, and removing software. It accomplishes this using a UNIX technology called containers."

You will learn how to use Docker to ship your web application to AWS in this chapter. Most importantly, we will show you how to build Docker images to bundle your web application. On top of that, you will study how to test your web application running inside containers locally.

Getting started with Docker

The most important part of the Docker toolkit is the container. A container is an isolated runtime environment preventing an application from accessing resources from other applications running on the same operating system. The concept of a jail - later called a container - had been around on UNIX systems for years. Docker uses the same ideas but makes them a lot easier to use.

The following figure illustrates the main difference between virtualization and containers. When using virtualization, the host operating system is managing access to the hardware resources. On top of that, the hypervisor simulates hardware for the guest operating systems. The applications are running inside the guest operating system. In contrast, Docker containers are sharing the same kernel. Therefore, starting a Docker container and accessing hardware resources is much more efficient.

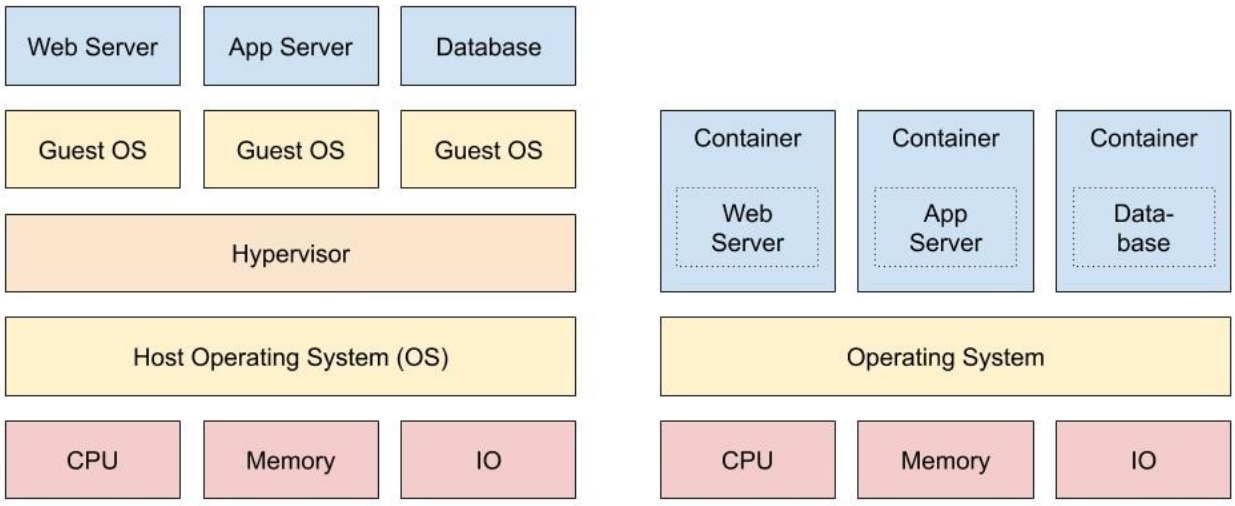


Figure 4: Virtual Machine vs. Container

Docker containers ship an application, including its runtime environment, to any operating system. As shown in the following figure, a Docker container includes binaries, configuration files, runtime environments, and 3rd party libraries. This allows you to run multiple applications on the same operating system without causing any side effects or conflicts. The Docker daemon creates and manages the containers.

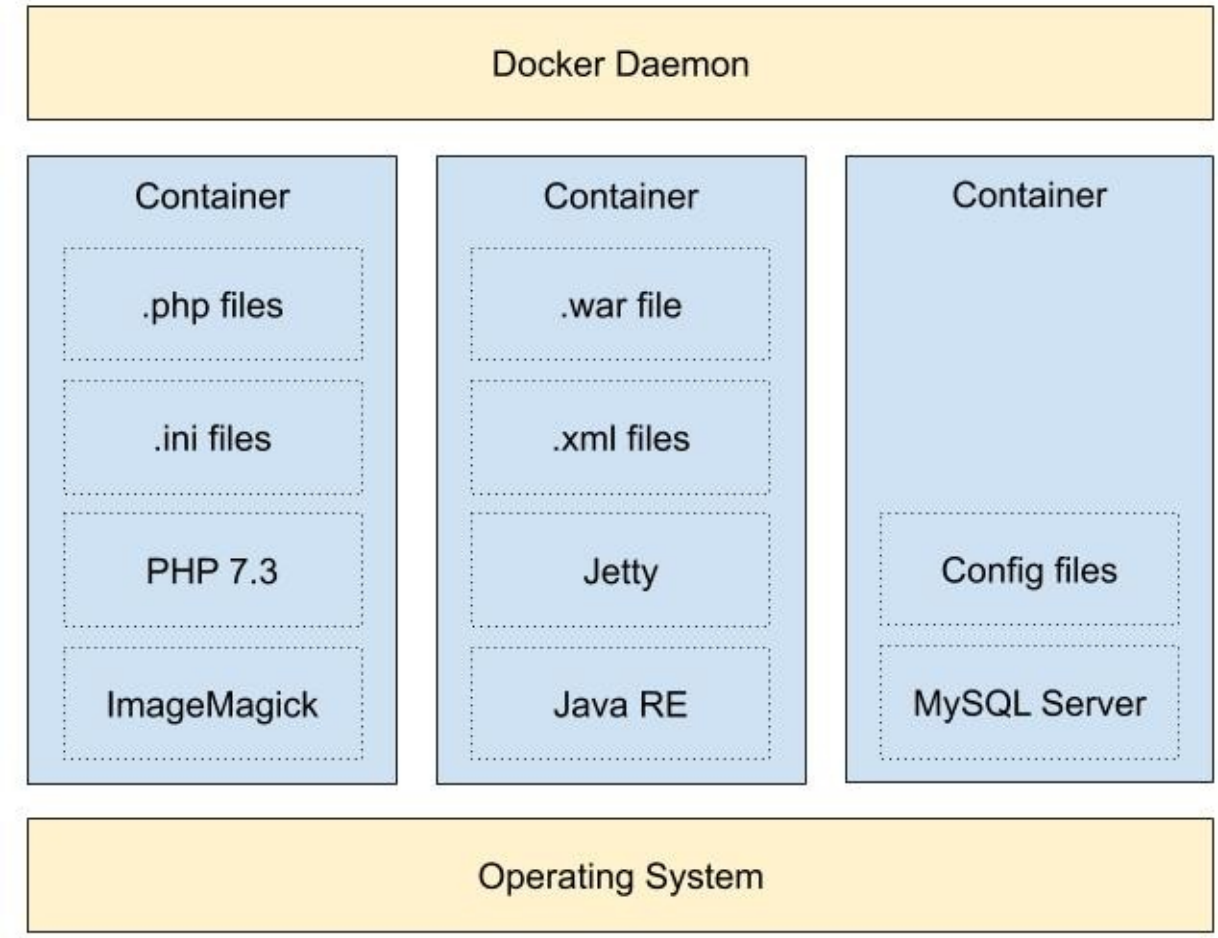


Figure 5: Multiple Container running on the same Operation System

Sound great? How do you start a Docker container on your machine? The following examples explain how to run a web server with Docker.

Execute the following command to start a Docker container running NGINX:

```
docker run -d -p 8080:80 nginx:1.14
```

If everything is working fine, the output should look like this:

```
Unable to find image 'nginx:1.14' locally
1.14: Pulling from library/nginx
27833a3ba0a5: Pull complete
0f23e58bd0b7: Pull complete
8ca774778e85: Pull complete
Digest: sha256:[...]df7563c3a2a6abe24160306b8d
Status: Downloaded newer image for nginx:1.14
419a3e9c025a3b2cc95fcf21e9721b4970ada7cf3e557fe498e7aa78b1cdaa25
```

You are running NGINX inside a container on your machine. Opening http://localhost:8080 returns the *Welcome to nginx!* page.

The following table explains the parameters of the `docker run` command to start a new container:

Parameter	Explanation
`-d`	Start in background mode.
`-p 8080:80`	Forward port 80 from the container to 8080 on your machine.
`nginx:1.14`	The name and version of the image, also known as a tag.

How do you list all the Docker containers running on your machine? The following command lists all running Docker containers.

```
docker ps
```

The output lists the running containers:

```
CONTAINER ID   IMAGE        COMMAND                  [...]
419a3e9c025a   nginx:1.14   "nginx -g 'daemon of"    [...]
```

To stop a container, type the following and replace `$CONTAINER_ID` with the ID of your container shown by the previous command:

```
docker stop $CONTAINER_ID
```

In my example, the following command stops the container:

```
docker stop 419a3e9c025a
```

The following command lists all containers, including the stopped ones:

```
docker ps -a
```

The container is stopped but not deleted. Use the following command to remove the container:

```
docker rm $CONTAINER_ID
```

In my example, the following command deletes the container:

```
docker rm 419a3e9c025a
```

Want to learn more? Visit the Docker run, ps, stop, and rm reference.

Before we proceed, I want to highlight a few essential best practices when working with Docker containers.

1. **Use one process per container.** If your application consists of more than one process, split them up into multiple containers. For example, if you run NGINX and PHP-FPM, create two containers. The PHP example in the following section shows how to do that.
2. **Do not use SSH.** Do not install or enable SSH within a container. Use `docker attach` to log into a container if needed for debugging. Or even better, optimize logging for debugging.
3. **Use environment variables instead of configuration files.** Do not use files to store the configuration for your application. Use environment variables instead. We will cover how to do so in the following section.
4. **Use standard output (stdout) and standard error (stderr) for logging.** Do not write log files. Docker has built-in support to ship log messages from STDOUT and STDERR to various centralized logging solutions.

Next, we will dockerize your application.

Building the Docker images for your web application

With Docker containers the differences between different platforms like your developer machine, your test system, and your production system are hidden under an abstraction layer. But how do you distribute your application with all its dependencies to multiple platforms? By creating a Docker image. A Docker image is similar to a virtual machine image, such as an Amazon Machine Image (AMI) that is used to launch an EC2 instance. The Docker image contains an operating system, the runtime environment, 3rd party libraries, and your application. The following figure illustrates how you can fetch an image and start a container on any platform.

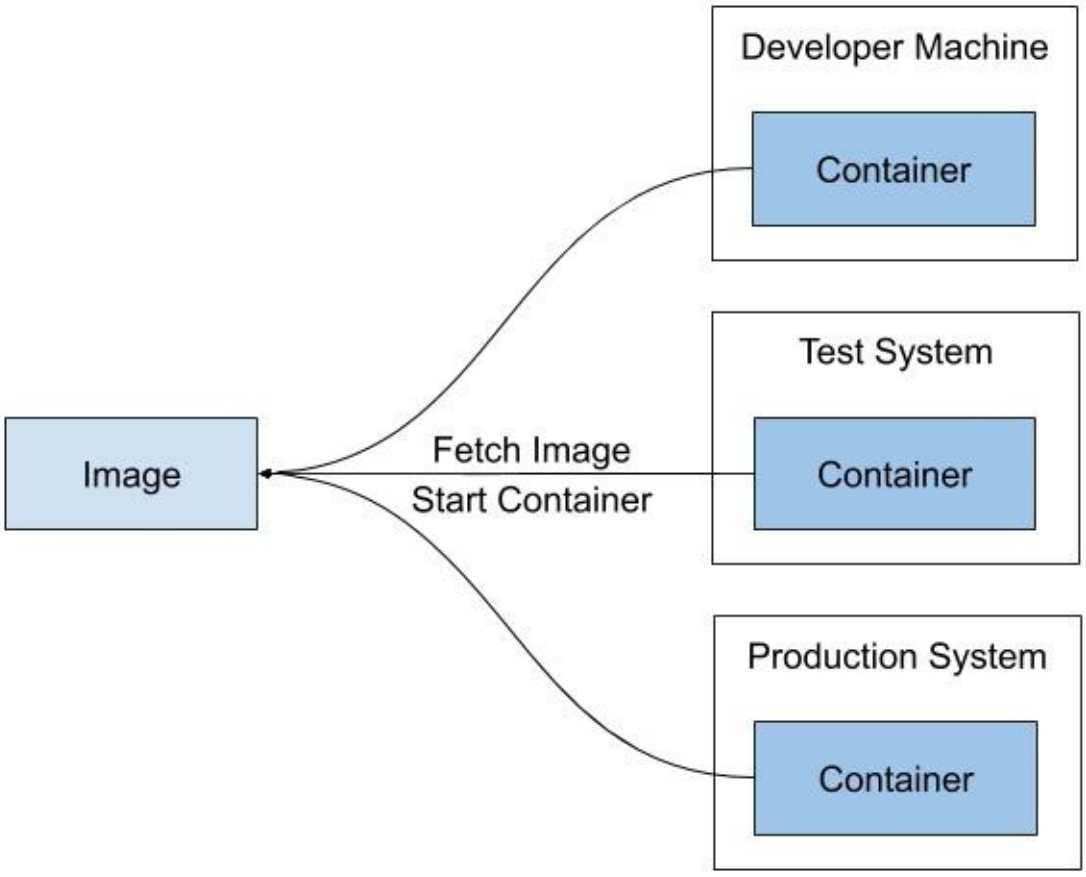


Figure 6: Distribute your application among multiple machines with an Docker image

But how do you create a Docker image for your web application? By creating a script that builds the image step by step: a so called `Dockerfile`.

In the following example you will learn how to dockerize your web application. The example uses a simple web application written in PHP without using any frameworks. You will find additional examples in the appendix at the end of the book.

> **Dependency** Before you proceed, please make sure you have followed "Installing Docker" and "Configuring your temporary working environment" as described in chapter 1.

A short reminder of how to start your temporary working environment:

On Mac and Linux

Open a Terminal and change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment:

```
bash start.sh
```

On Windows

Open PowerShell and change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment:

```
powershell -ExecutionPolicy ByPass -File start.ps1
```

Now that you have started the temporary working environment on your local machine (MacOS, Linux, or Windows), execute the following command to change directory:

```
cd /src/php-basic
```

Which includes our sample application:

```
ls
```

- `conf` the configuration directory (contains .ini files)
- `css` the stylesheet directory (contains .css files)
- `img` the images directory (contains .gif files)
- `lib` the libraries directory (contains .php files)
- `index.php` the main file

We have already added two additional folders to dockerize and launch the sample application:

- `docker` the directory containing the Docker configuration (e.g., the `Dockerfile`)
- `aws` the directory containing the templates to deploy to AWS

When following the next steps, it is up to you whether you are using an editor - vi or nano - to show and edit the files from within your temporary working environment, or if you prefer opening the files from your disk with an editor of your choice. You do not need to create or edit any files in the following part of the chapter if you just want to start our sample PHP application. If you want to dockerize your own PHP application, watch for the customization boxes which describe how you might adapt the code to your needs.

> **Customization** A box indicating that you might need to customize something here in your project.

A typical setup to serve a PHP application consists of:

1. A web server (for example NGINX)
2. A PHP process (for example PHP-FPM)

Therefore, we need to run two processes: NGINX and PHP-FPM. However, a container should only run exactly one process at a time. Which means we need to build two images. The following figure shows the two containers: the NGINX container receives the request from the client and forwards PHP requests to the PHP-FPM container. Both containers run on the same host to avoid additional network latency.

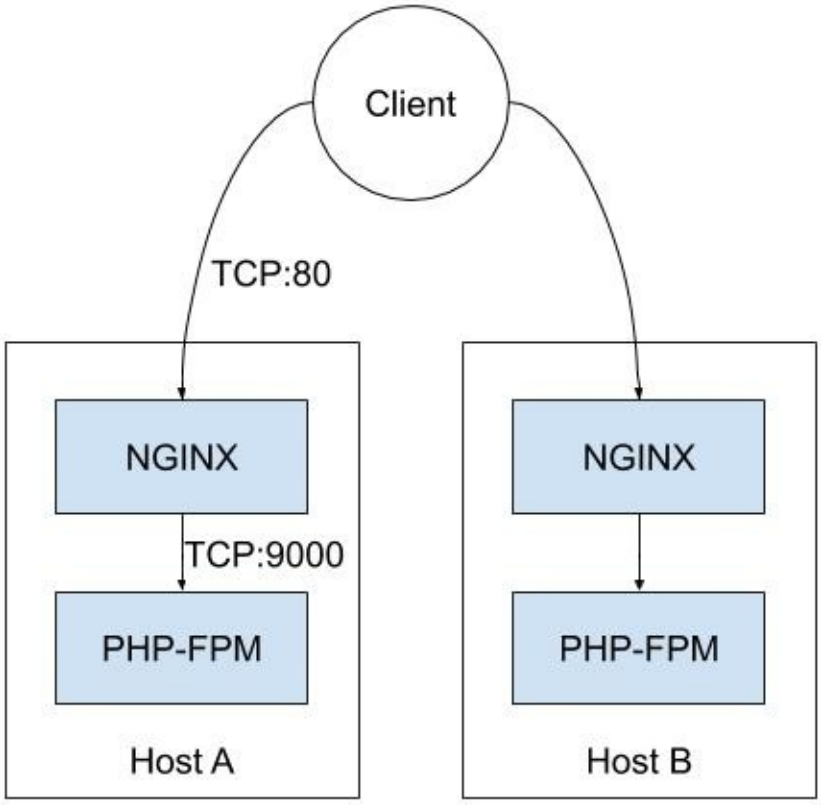


Figure 7: Proxy pattern: NGINX and PHP-FPM containers running on the same machine

Start with creating a Docker image for NGINX. NGINX serves the static files. In our example application the static files are stored in the `css` and `img` directory already. On top of that, NGINX forwards PHP requests to PHP-FPM.

The following snippet shows the configuration file `docker/nginx/default.conf` which tells NGINX to serve static files from `/var/www/html` and forward PHP requests to PHP-FPM. You do not need to make any changes to the NGINX configuration when dockerizing your web application.

```
server {
    listen       80;
    server_name  localhost;
    root         /var/www/html;
    # pass the PHP scripts to FastCGI server
    # listening on 127.0.0.1:9000
    location ~ \.php$ {
        fastcgi_pass   127.0.0.1:9000;
        fastcgi_index  index.php;
        fastcgi_param  SCRIPT_FILENAME  $document_root$fastcgi_scrip
```

```
        fastcgi_param  SCRIPT_NAME      $fastcgi_script_name;
        include        fastcgi_params;
    }
    # redirect / to index.php
    location ~ ^\/$ {
        return 301 $scheme://$http_host/index.php$is_args$args;
    }
}
```

Next, you need a `Dockerfile` for building your own NGINX image. The following snippet shows the file `docker/nginx/Dockerfile` that we created for our sample application.

The first instruction defines the base image. When creating an image, we don't have to start from scratch. We can use a pre-built base image.

```
FROM nginx:1.14
```

The next instruction copies the NGINX configuration file from your disk to the Docker image.

```
COPY docker/nginx/default.conf /etc/nginx/conf.d/default.conf
```

The next two instructions copy the `css` and `img` directories from your local disk to the NGINX root directory `/var/www/html/` in the Docker image.

> **Customization** Depending on where you are storing the static files of your web application, you'll need to modify these instructions accordingly. Make sure you are copying all static files to `/var/www/html/`.

```
COPY css /var/www/html/css
COPY img /var/www/html/img
```

The next instruction runs the chown command to transfer ownership of all static files to the `nginx` user. The nginx user is part of the base image.

```
RUN chown -R nginx:nginx /var/www/html
```

The `Dockerfile` is ready. It's time to build your first image. Build the image from within your working environment container that you have started at the beginning of section 2.2.

```
cd /src/php-basic/
```

```
docker build -t php-basic-nginx:latest -f docker/nginx/Dockerfile .
```

The following table explains the parameters of the `docker build` command to build a new image:

Parameter	Explanation
`-t php-basic-nginx:latest`	Add a tag (name) to the new image.
`-f docker/nginx/Dockerfile`	Location of the Dockerfile.
`.`	Use the current directory as the build context (all paths, e.g. in COPY, are relative to the build context).

The next step is building the PHP-FPM image. The following snippets show the file `docker/php-fpm/Dockerfile` used by our sample application.

The first instruction defines the base image. We are using a base image with PHP 7.3 pre-installed for our sample application.

Customization The following versions are supported as well: 7.2 and 7.1.

```
FROM php:7.3-fpm-stretch
```

Followed by enabling the PHP configuration optimized for production workloads and installing the PHP extensions `pdo` and `pdo_mysql`.

Customization Feel free to install additional extensions if needed.

```
RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"
RUN docker-php-ext-install -j$(nproc) pdo pdo_mysql
```

The next commands install `envsubst`. You will learn more about how to create configuration files with `envsubst` in a second. No need to change anything here.

```
RUN apt-get update && apt-get install -y gettext
COPY docker/php-fpm/custom-entrypoint /usr/local/bin/
RUN chmod u+x /usr/local/bin/custom-entrypoint
ENTRYPOINT ["custom-entrypoint"]
RUN mkdir /var/www/html/conf/
```

Afterwards, the script copies all configuration templates `.tmp` located in `conf/`.

```
COPY conf/*.tmp /tmp/conf/
```

The following instructions copy the PHP files from your disk to the root directory of PHP-FPM.

> **Customization** When dockerizing your own application you will most likely need to modify these COPY instructions to make sure all PHP files are copied to the image. Also, as you add new PHP files to your application, make sure to add them to this list as well.

```
COPY *.php /var/www/html/
COPY lib /var/www/html/lib
RUN chown -R www-data:www-data /var/www/html
```

The last instruction tells Docker to start the PHP-FPM process by default. You do not need to change anything here.

```
CMD ["php-fpm"]
```

The `Dockerfile` is ready. But we have skipped one challenge: the configuration files. We assume you are storing the configuration for your web application within files. When using Docker, and especially when using Fargate, using configuration files is cumbersome. Instead, you should use environment variables to configure your application.

You have two options:

- Modify your application to read all configuration from environment variables.
- Do not modify your application, but use environment variables to create configuration files with `envsubst`.

In the following steps you will learn how to write configuration files on container startup based on environment variables with `envsubst`.

Our sample application uses a configuration file to configure the database connection: `conf/app.ini`

```
[database]
host=mysql
name=test
user=app
```

```
password=secret
```

Our goal is to use environment variables for each property. To do so with `envsubst`, we need to create a configuration template. The following snippet shows the template `conf/app.ini.tmp` for our `conf/app.ini` configuration file.

In the template, each value has been replaced with a placeholder. For example, `${DATABASE_HOST}` references the environment variable `DATABASE_HOST`.

```
[database]
host="${DATABASE_HOST}"
name="${DATABASE_NAME}"
user="${DATABASE_USER}"
password="${DATABASE_PASSWORD}"
```

By default, the `Dockerfile` adds all configuration template files `.tmp` stored in `conf` to the image. Each time a container starts it executes the script located in `docker/php-fpm/custom-entrypoint`. The following snippet shows how the `custom-entrypoint` script creates the configuration files based on all your configuration templates.

```
echo "generating configuration files"
FILES=/tmp/conf/*
for f in $FILES
do
  c=$(basename $f .tmp)
  echo "... $c"
  envsubst < $f > /var/www/html/conf/${c}
done
```

> **Customization** To add your own configuration files, create a configuration template by replacing all dynamic values with placeholders (e.g. `$ENV_NAME`). Store the configuration template in the `conf` folder. That's it. During runtime the container will create a configuration file based on the template.

Next, use the `docker build` command to create your PHP-FPM image:

```
docker build -t php-basic-php-fpm:latest \
  -f docker/php-fpm/Dockerfile .
```

You have successfully built two Docker images. In the following sections you will learn how to test your web application locally, push your images to a private

registry, as well as how to launch your web application on AWS.

Testing your web application locally

As promised one of the benefits of Docker is that you can test your application locally. To do so, you need to spin up three containers:

- NGINX
- PHP-FPM
- MySQL (which will be replaced by Amazon Aurora when deploying on AWS)

Theoretically, you can create the needed containers manually with `docker run` but doing so is a little cumbersome. You will use Docker Compose, a tool for running multi-container applications, instead.

The following snippet shows the Docker Compose file `docker-compose.yml` located in the `docker` folder of the `php-basic` sample application.

> **Customization** You probably need to add your own environment variables for the PHP container (see inline comments).

```
version: '3'
services:
  nginx:
    build:
      context: '..'
      dockerfile: 'docker/nginx/Dockerfile' # build your NGINX image
    depends_on:
    - php
    network_mode: 'service:php' # use network interface of php conta
  php:
    build:
      context: '..'
      dockerfile: 'docker/php-fpm/Dockerfile' # build PHP image
    ports:
    - '8080:80' # forwards port of nginx container
    depends_on:
    - mysql
    environment: # add your own variables used by envsubst here
      DATABASE_HOST: mysql
      DATABASE_NAME: app
      DATABASE_USER: app
      DATABASE_PASSWORD: secret
  mysql:
```

```
    image: 'mysql:5.6' # matches the Amazon Aurora MySQL version
    command:'--default-authentication-plugin=mysql_native_password'
    ports:
    - '3306:3306' # forwards port 3306 to 3306 on your machine
    environment:
      MYSQL_ROOT_PASSWORD: secret # password for root user
      MYSQL_DATABASE: app # create database with name app
      MYSQL_USER: app # user app is granted full access to db app
      MYSQL_PASSWORD: secret # the password for user app
```

From within your temporary working environment execute the following command to spin up the containers on your machine.

```
docker-compose -f docker/docker-compose.yml up
```

Magically, Docker Compose will spin up three containers. Point your browser to http://localhost:8080/index.php to check that your web application is up and running. The log files of all containers will show up in your terminal which simplifies debugging a lot.

If you need to make a change to your setup, please cancel the running `docker-compose` process `CTRL + C` and restart with the following command afterwards to make sure the images get re-built.

```
docker-compose -f docker/docker-compose.yml up --build
```

Use your favorite MySQL client and connect to `localhost:3306` with username `root` and password `secret` if you need to create a schema or restore a database dump.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers.

```
docker-compose -f docker/docker-compose.yml down
```

It's time to deploy your web application on AWS.

Pushing your Docker image to the Amazon ECR registry

After building and testing an image, how do you share it with others or distribute it among different platforms? A Docker registry offers a convenient way to store and distribute Docker images. As shown in the following figure, you can push an image to the registry from your developer machine and pull the image from the test or production system. Archiving different versions of your Docker image is another key part of a Docker registry.

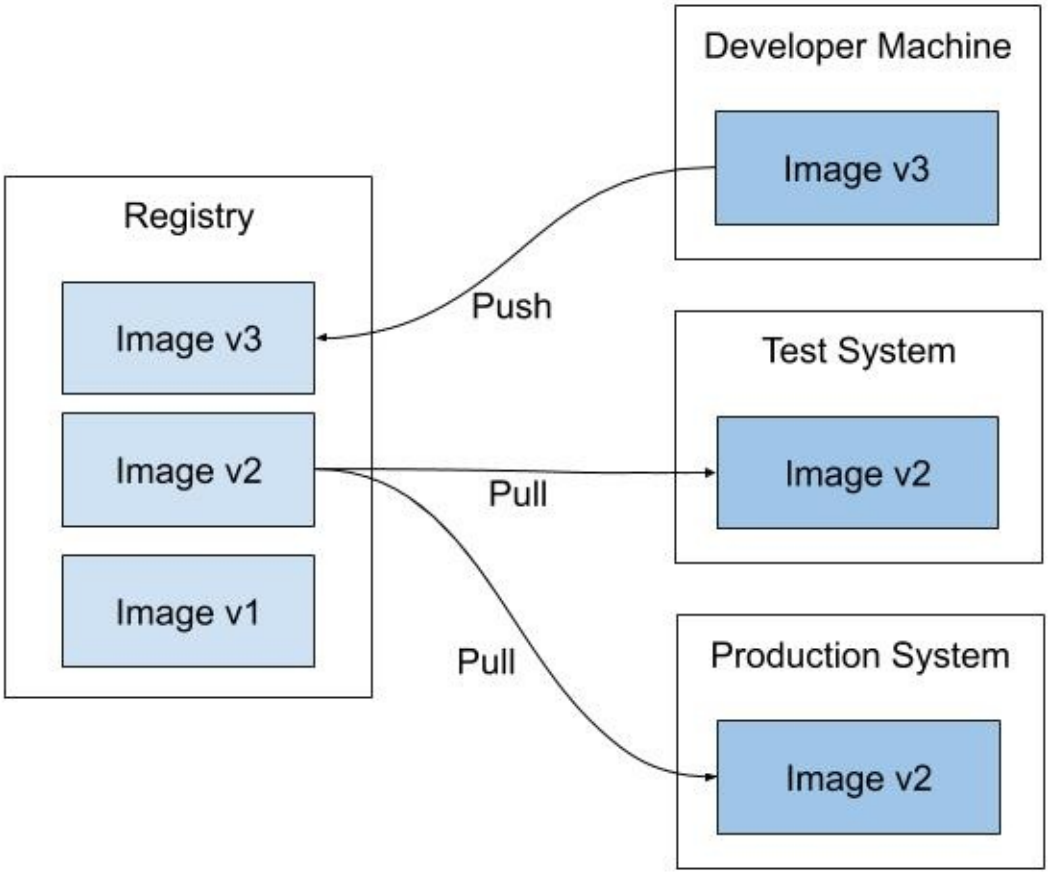


Figure 8: Docker registry: push and pull images

The most popular Docker registry is Docker Hub. You have fetched the base images from Docker Hub.

AWS offers a Docker registry as well: Amazon Elastic Container Registry

(ECR). We recommend using ECR as it is easy to use, private, billed per usage and tightly integrated with ECS and Fargate.

First of all, you need to create an ECR repository for your NGINX images.

```
aws ecr create-repository --repository-name php-basic-nginx \
  --query 'repository.repositoryUri' --output text
```

The command returns a repository URI. Please take note of the URI which looks similar to the following output.

```
111111111111.dkr.ecr.eu-west-1.amazonaws.com/php-basic-nginx
```

Next, create an ECR repository to store your PHP-FPM images.

```
aws ecr create-repository --repository-name php-basic-php-fpm \
  --query 'repository.repositoryUri' --output text
```

Again, please note down the repository URI which looks similar to the following output.

```
111111111111.dkr.ecr.eu-west-1.amazonaws.com/php-basic-php-fpm
```

Next, you need to login to be able to access your ECR repositories from the Docker engine:

```
$(aws ecr get-login --no-include-email)
```

To push your NGINX image to ECR, you need to tag and push the image. Replace `$REPOSITORY_URI_NGINX` with the first output from above.

```
docker tag php-basic-nginx:latest $REPOSITORY_URI_NGINX:latest
docker push $REPOSITORY_URI_NGINX:latest
```

Which should look similar to the following example:

```
docker tag php-basic-nginx:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-basic-nginx:latest
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-basic-nginx:latest
```

To push your PHP-FPM image to ECR, you need to tag and push the image.

Replace `$REPOSITORY_URI_PHP_FPM` with the second output from above.

```
docker tag php-basic-php-fpm:latest $REPOSITORY_URI_PHP_FPM:latest
docker push $REPOSITORY_URI_PHP_FPM:latest
```

For me, the commands looked as follows.

```
docker tag php-basic-php-fpm:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-basic-php-fpm:latest
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-basic-php-fpm:latest
```

Use the following commands to verify that your images are stored in your repositories.

```
aws ecr list-images --repository-name php-basic-nginx
```

The output should look something like this:

```
{
  "imageIds": [
    {
      "imageTag": "latest",
      "imageDigest": "sha256:b7df23c0800aaaeb5006954d528[...]"
    }
  ]
}
```

Repeat for the PHP-FPM repository.

```
aws ecr list-images --repository-name php-basic-php-fpm
```

The next section is about launching your web application on AWS.

Launching your web application

How to configure your web application when running on AWS? We will guide you through the configuration of our example application. Open the template `aws/template.yml` used to deploy your application to AWS, please. Search for the resource named `Service` which is shown in the following snippet.

```
Service:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      # [...]
      AppEnvironment1Key: 'DATABASE_PASSWORD'
      AppEnvironment1SecretModule: !GetAtt 'Secret.Outputs.StackName
      AppEnvironment2Key: 'DATABASE_HOST'
      AppEnvironment2Value:
        !GetAtt 'AuroraServerlessCluster.Outputs.DnsName'
      AppEnvironment3Key: 'DATABASE_NAME'
      AppEnvironment3Value: 'test'
      AppEnvironment4Key: 'DATABASE_USER'
      AppEnvironment4Value: 'master'
```

You will find two parameters for each numbered environment variable:

- `AppEnvironment1Key` the key aka. name of environment variable 1.
- `AppEnvironment1Value` the plain-text value of environment variable 1 or `AppEnvironment1SecretModule` to pass a secret.

Customization Have you added or edited the environment variables to work with your own application in 2.2? Then please don't forget to to modify the key and the value for each of the environment variables in `aws/template.yml` as well.

One more thing: in our example, we need to pass the hostname (DNS name) of the database to the application. The `!GetAtt` inserts the DNS name of the database `AuroraServerlessCluster.Outputs.DnsName` automatically. A similar mechanism is used to access a generated database password: Instead of a plain-text value you pass the name of the module containing the secret via `AppEnvironment1SecretModule`.

To deploy your newly dockerized web application, execute the following

commands in your temporary working environment:

- Replace `$NICKNAME` with the nickname you chose when setting up your temporary working environment in chapter 1.
- Replace `$NGINX_IMAGE` with your NGINX image (e.g., `111111111111.dkr.ecr.eu-west-1.amazonaws.com/php-basic-nginx:latest`).
- Replace `$PHP_IMAGE` with your PHP image (e.g., `111111111111.dkr.ecr.eu-west-1.amazonaws.com/php-basic-php-fpm:latest`).

```
cd /src/php-basic/aws
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
aws cloudformation deploy --template-file .template.yml \
  --stack-name php-basic --capabilities CAPABILITY_IAM \
  --parameter-overrides AppImage=$PHP_IMAGE \
ProxyImage=$NGINX_IMAGE
```

Provisioning your AWS infrastructure will take up to 15 minutes.

Use the following command to retrieve the URL to your web application running on AWS and open it in your browser:

```
aws cloudformation describe-stacks --stack-name php-basic \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

In case you are not planning to continue with the next chapter in a timely manner, please delete the AWS infrastructure and repositories to avoid unnecessary costs. Otherwise, keep the infrastructure up and running.

```
cd /src/php-basic/aws
bash cleanup.sh
```

Is your web application not working? We will discuss how to monitor and debug it in the next chapter. Need help? Please share your problem with us and the community: https://community.cloudonaut.io.

Mastering the building blocks of the cloud infrastructure

So far, you have launched your web application on AWS sans dealing with any details of the underlying cloud infrastructure. We hope you enjoyed the “rapid” experience.

In this chapter, you will learn the building blocks of CloudFormation, ALB, ECS, Fargate, RDS, and CloudWatch in detail. You will also dive into monitoring and debugging. On top of that, this chapter covers using a custom domain and an SSL certificate. Finally, you will learn how to run scheduled jobs.

Managing your stack with infrastructure as code: AWS CloudFormation

The secret sauce allowing you to set up your cloud infrastructure within minutes, which consists of around 100 resources, is Infrastructure as Code. It removes the need for clicking through the AWS Management Console manually.

In our opinion, Infrastructure as Code works best when following the declarative approach. Define the target state in source code and use a tool which calculates and executes the needed steps to transform the current state into the target state. Our tool of choice is AWS CloudFormation.

The figure below explains the key concepts of CloudFormation.

- You create a JSON or YAML file describing the target state of your infrastructure, which CloudFormation calls a **template**.
- You upload your template and ask CloudFormation to create, update or delete your infrastructure. The state of your infrastructure is stored within a so-called **stack**.
- CloudFormation transforms the current state of your stack into the target state defined in your template. To do so, CloudFormation creates, updates, or deletes **resources** as needed.

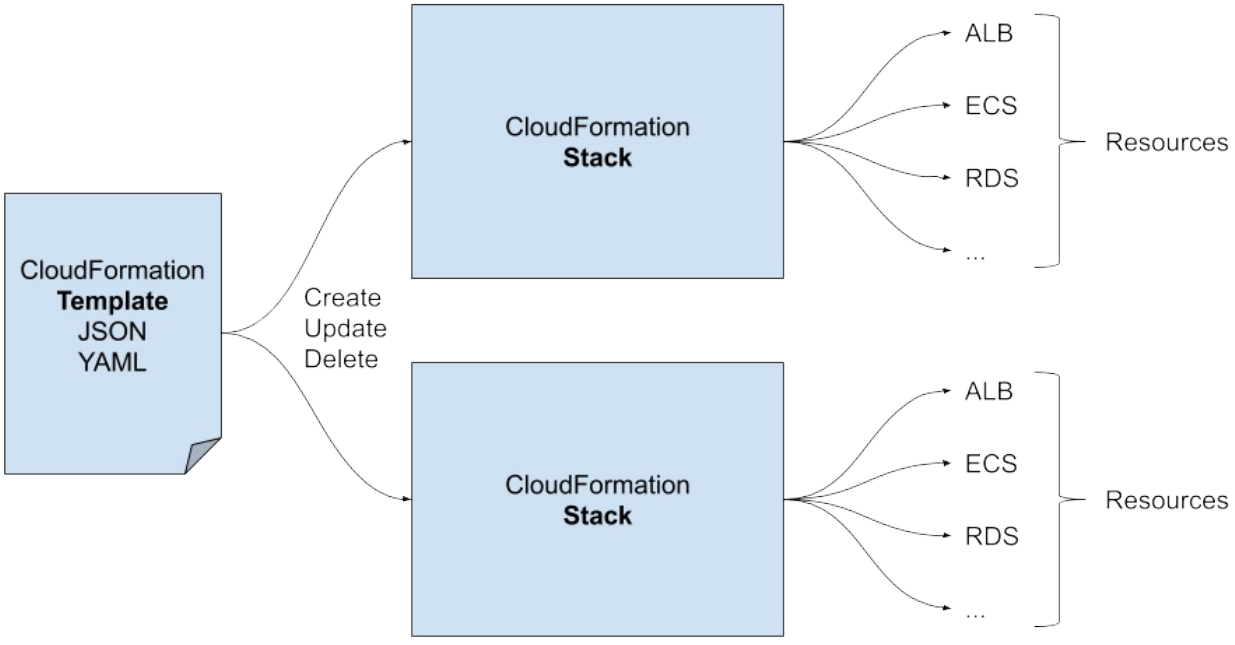


Figure 9: Infrastructure as Code with CloudFormation: Use a template to create a new stack which creates the resources

The first step when using CloudFormation is to create a template. As mentioned before, CloudFormation understands templates written in JSON and YAML. We recommend using YAML. Haven't worked with YAML before? Check out the Grav's YAML Syntax introduction. The snippet below shows a basic template.

- It is a convention to begin a YAML file with `---`.
- We highly recommend specifying the template format version with `AWSTemplateFormatVersion`. Currently, the only available version is `2010-09-09`.
- The `Resources` section is the most important part of the template as it includes a description of the target state.
- The template includes a single resource with the **logical id** `Cluster` of **type** `AWS::ECS::Cluster`. The resource describes an ECS cluster and specifies the cluster name with the **property** `ClusterName`.
- Please note, the whitespaces used to indent the lines are part of the YAML syntax.

```
---
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  Cluster:
    Type: 'AWS::ECS::Cluster'
    Properties:
      ClusterName: 'my-cluster'
```

CloudFormation supports a nearly endless list of AWS resources. Check out the AWS Resource and Property Types Reference to learn more.

Let's extend the basic template with a second resource.

- The template now includes two resources with the logical ids `Cluster` and `Service`.
- The second resource is of type `AWS::ECS::Service`, specifying an ECS service.
- The `Service` resource defines three properties: `Cluster`, `DesiredCount`, and `LaunchType`.
- The value for the property `Cluster` for `Service` is not hard-coded. Instead, the value references the name of the resource `Cluster` by using the

intrinsic function `!Ref`.

```
---
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  Cluster:
    Type: 'AWS::ECS::Cluster'
    Properties:
      ClusterName: 'my-cluster'
  Service:
    Type: 'AWS::ECS::Service'
    Properties:
      Cluster: !Ref Cluster
      DesiredCount: 2
      LaunchType: FARGATE
      # [...]
```

It is not uncommon for a template to grow to more than 1,000 lines of code. Modularizing your template into smaller parts is essential to keeping your Infrastructure as Code readable. Therefore, we launched an open-source project called cfn-modules, providing production-ready modules to build your cloud infrastructure. Next, you will learn how to use cfn-modules, as we are heavily using the modules within this book.

> **Dependency** Before you proceed, please make sure you have followed "Installing Docker" and "Configuring your temporary working environment" as described in chapter 1.

A short reminder of how to start your temporary working environment:

On Mac and Linux

Open a Terminal and change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment:

```
bash start.sh
```

On Windows

Open PowerShell and change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment:

```
powershell -ExecutionPolicy ByPass -File start.ps1
```

Now that you have started the temporary working environment on your local machine (macOS, Linux, or Windows), execute the following command to change directory:

```
cd /src/php-basic/aws
```

The figure below illustrates how to use modules to put together your infrastructure.

- The `template.yml` file describes the parent stack.
- The parent stack consists of several child stacks.
- The parent stack manages the dependencies between the child stacks.

CloudFormation calls this approach **nested stacks**. See Working with Nested Stacks to learn more.

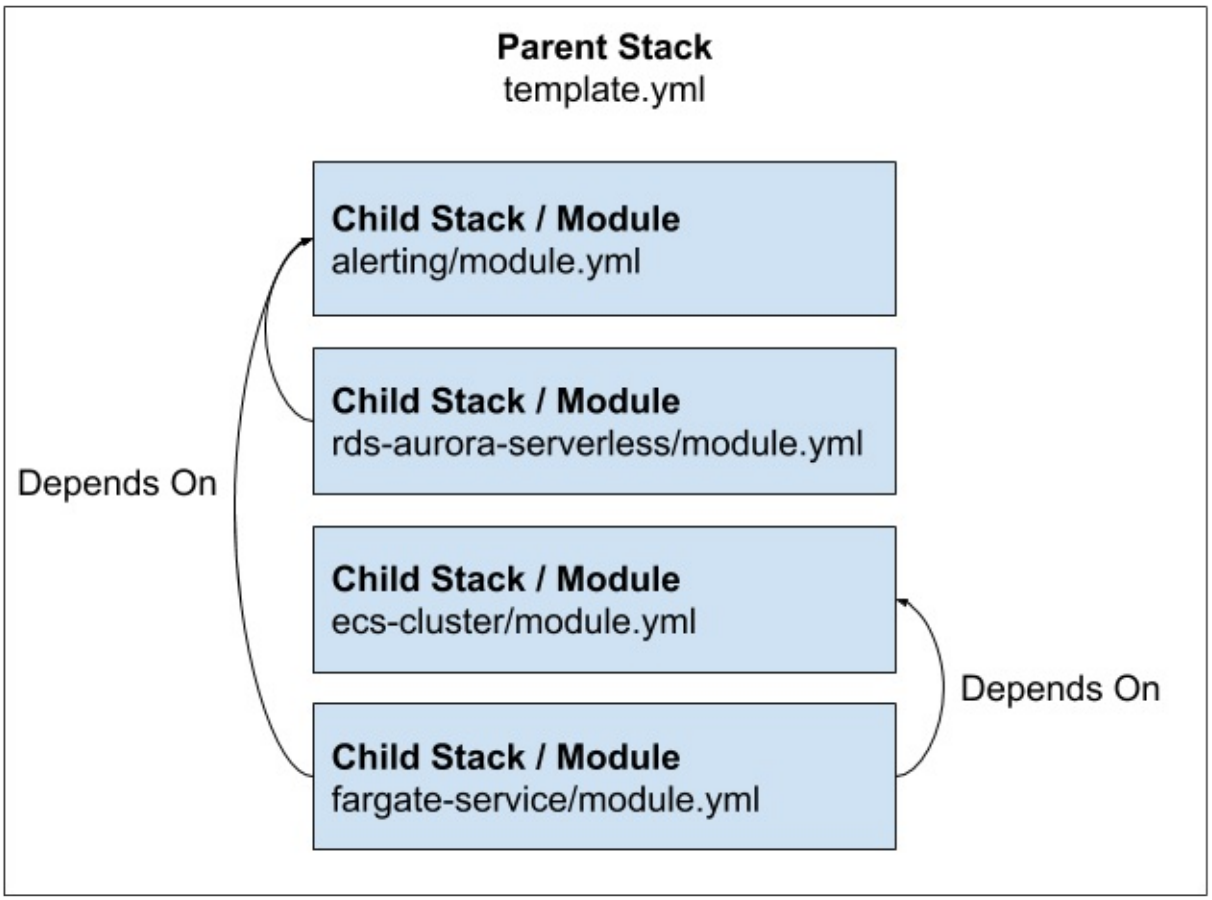


Figure 10: Nested Stacks with cfn-modules

Whenever you want to modify your cloud infrastructure, open the template of the parent stack. The snippet below illustrates how the modules are wired together.

- The snippet includes four modules named `Alerting`, `AuroraServerlessCluster`, `Cluster`, and `AppService`.
- Each module resource is of type `AWS::CloudFormation::Stack`, which tells CloudFormation to create a nested stack.
- The property `TemplateURL` of each module resource references a CloudFormation template.
- The property `Parameters` defines the parameters for each module.
- There are dependencies between the modules. For example, `AuroraServerlessCluster` depends on `Alerting`. The name of the `Alerting` nested stack is handed over to the `AuroraServerlessCluster` nested stack: `AlertingModule: !GetAtt 'Alerting.Outputs.StackName'`.

```
---
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  Alerting:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      Parameters:
        # [...]
      TemplateURL: '[...]'
  AuroraServerlessCluster:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      Parameters:
        AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
        # [...]
      TemplateURL: '[...]'
  Cluster:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      TemplateURL: '[...]'
  AppService:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      Parameters:
        ClusterModule: !GetAtt 'Cluster.Outputs.StackName'
        AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
        # [...]
      TemplateURL: '[...]'
```

Next, you will increase the CPU and memory capacity of your containers. To do so, find the module named AppService and replace the value of the parameter `Cpu` and `Memory`.

```
---
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  AppService:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      Parameters:
        ClusterModule: !GetAtt 'Cluster.Outputs.StackName'
        AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
        ProxyImage: !Ref ProxyImage
        ProxyPort: '80'
        Cpu: '1' # Increasing CPU to 1 vCPU
        Memory: '2' # Increasing Memory to 2 GB
      TemplateURL: './[...]/fargate-service/module.yml'
```

Does the `fargate-service` module offer any other parameters for customization? Find out!

1. Look for the name of the cfn-module. It is coded into the module resource's `TemplateURL`. The name of the module is `fargate-service`, extracted from `./node_modules/@cfn-modules/fargate-service/module.yml`.
2. Open the module's documentation cfn-modules/fargate-service on GitHub.
3. The documentation includes a list of all supported parameters.
4. If needed, add a parameter to the module resource.

The following shows how to add the parameter `HealthCheckGracePeriodSeconds`, which is needed if your application needs more than 60 seconds to start, for example:

```
---
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  AppService:
    Type: 'AWS::CloudFormation::Stack'
    Properties:
      Parameters:
        ClusterModule: !GetAtt 'Cluster.Outputs.StackName'
        AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
        ProxyImage: !Ref ProxyImage
        ProxyPort: '80'
        Cpu: '0.25'
        Memory: '0.5'
        HealthCheckGracePeriodSeconds: '120' # Adding a parameter
      TemplateURL: './[...]/fargate-service/module.yml'
```

You will learn how to install new modules while going through the rest of this chapter (see 3.5 and 3.7). The modules provided by cfn-modules do not limit you. Add any available CloudFormation resources to your template template.yml when needed.

> **Warning** Do not touch resources managed by CloudFormation. You should only use CloudFormation to modify or delete those resources. Do not use the AWS Management Console to apply modifications to your cloud infrastructure manually. Otherwise, your manual changes might be overridden in the future. Or even worse, CloudFormation will not be able to make any changes to your stack in the future.

After you are familiar with CloudFormation and the cfn-modules, it is time to

have a more in-depth look at the different building blocks of the "Rapid Docker on AWS" architecture.

Load-balancing requests to your containers: Amazon ALB

The Elastic Container Service (ECS) creates and deletes containers constantly: during deployments, when recovering from a failure, or scaling the number of containers depending on the current workload. When using Fargate, each container starts with a static port but a dynamic IP address. So how does a client connect to a container? The easiest way to do so is by using a load balancer. As shown in the figure below, the client connects to the load balancer (ALB) by resolving its static DNS name (e.g., `alb.us-east-1.elb.amazonaws.com`). The client sends an HTTP request to the load balancer's IP address. Next, the load balancer forwards the request to one of the available and healthy containers.

Typically, the ALB is the only part of your infrastructure allowing incoming traffic from the Internet. The communication between the ALB and the containers happens within a private network called a VPC.

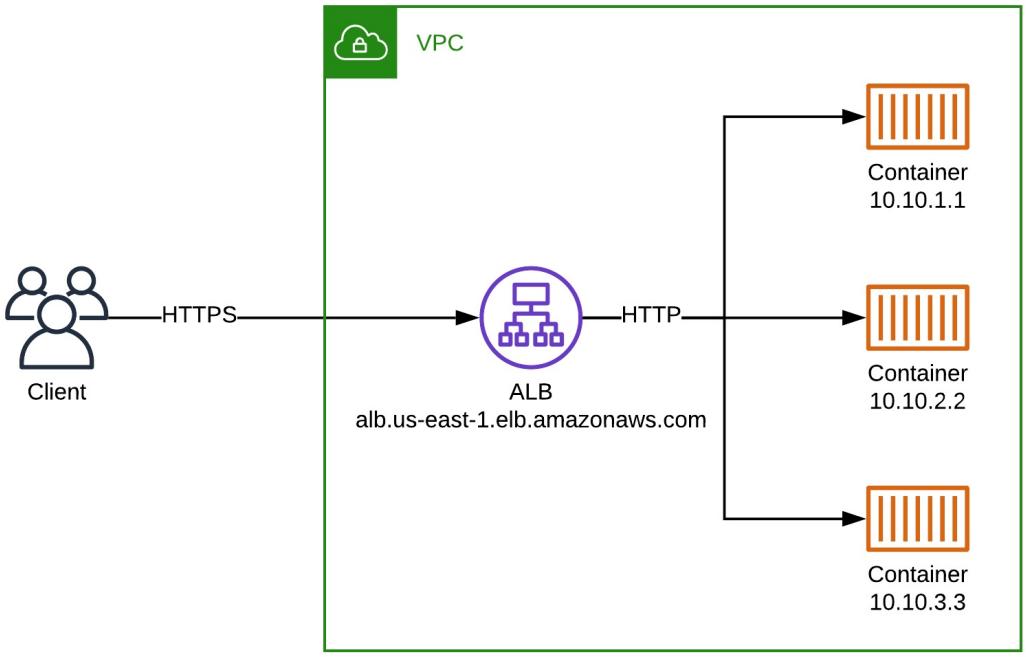


Figure 11: The ALB accepts requests from clients and forwards them to a container

AWS offers different types of load balancers. We are using the Application Load Balancer (ALB). The ALB operates at layer 7, also called the application layer. This means that the ALB supports HTTP and HTTPS and uses rules to forward requests based on the HTTP method, headers, query string, hostname, and source IP address. Please note, to enable HTTPS you need to follow the instructions in section 3.4.

The figure below illustrates the main components of an ALB.

- The **ALB** is the core component.
- A **Target Group** is a collection of targets (e.g., containers, virtual machines, ...).
- A **Listener** listens for incoming HTTP or HTTPS requests on a specific port. Typically, an ALB listens on port 80 (HTTP) and 443 (HTTPS).
- A **Rule** contains a condition (e.g., the path begins with `/static/`) and references a target group.

Let's track a request from the client to a container:

1. The client sends an HTTPS request with path `/static/img.png`.
2. The request arrives at the HTTPS listener of the ALB.
3. The ALB evaluates the rules attached to the HTTPS listener.
4. There is a match with the Rule `/static/`.
5. The ALB sends an HTTP request to one of the containers registered in target group A.
6. The container answers the HTTP request.
7. The ALB receives the response.
8. The ALB sends the response to the client.

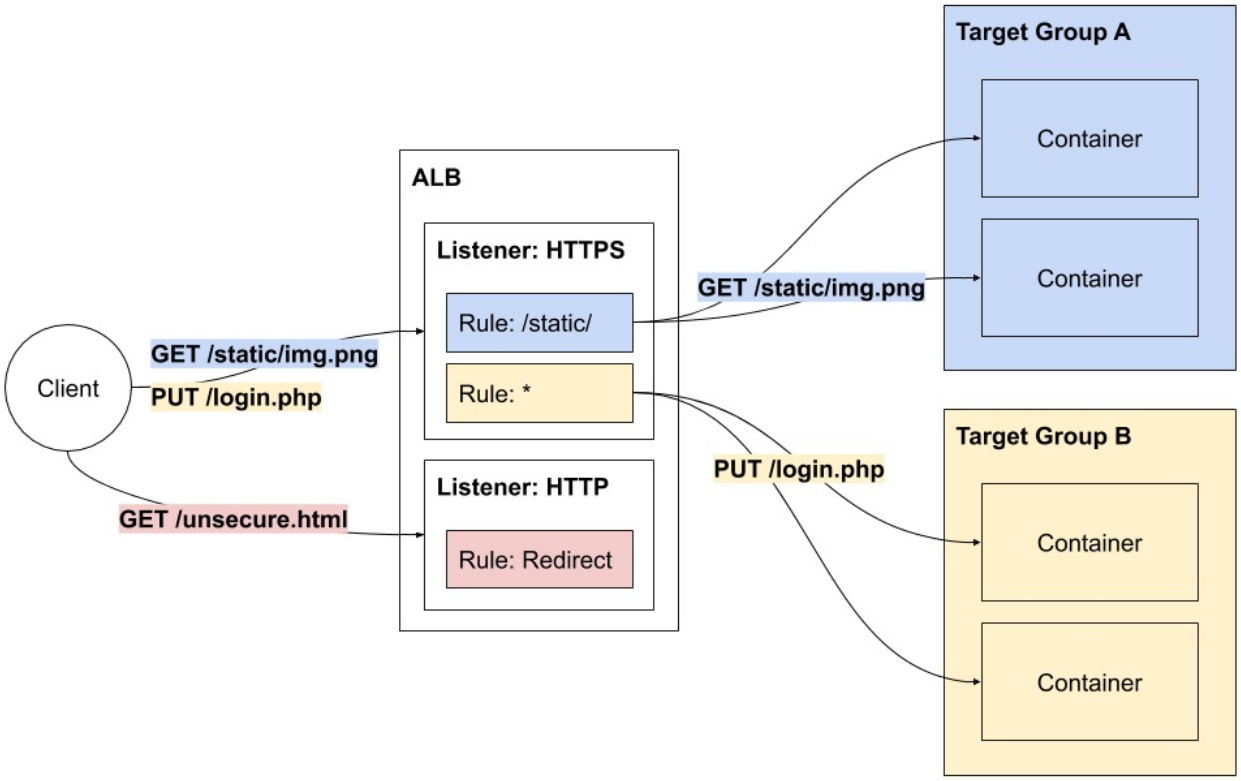


Figure 12: Explaining the ALB concepts: target group, listener, and rule

Besides routing requests based on rules, the ALB is also constantly checking the health of all targets. To do so, the ALB sends HTTP requests to each container. As shown in the following figure, the ALB forwards incoming requests to healthy containers only.

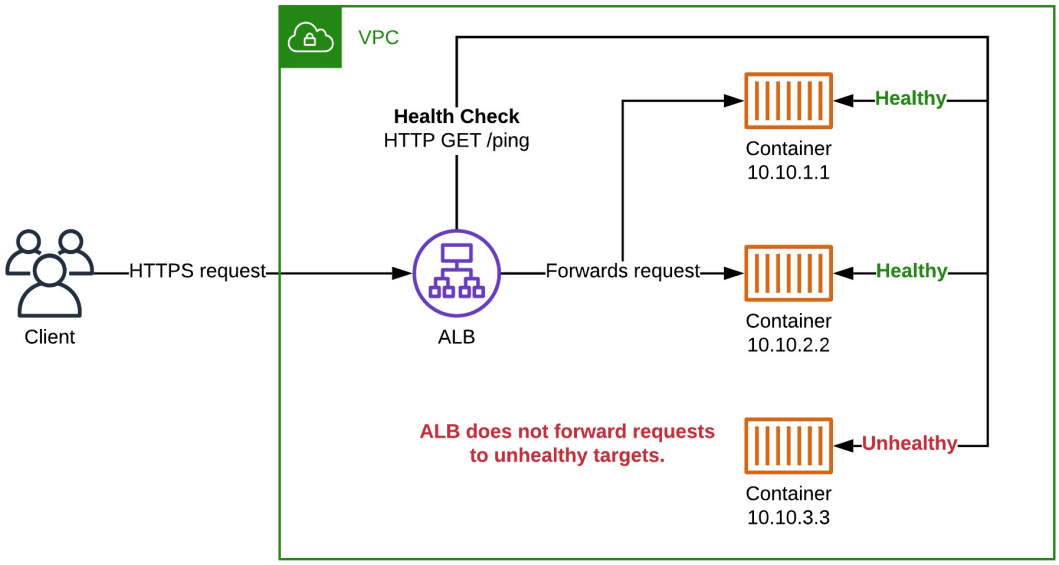


Figure 13: The ALB does not forward requests to unhealthy targets

Our modules configure a health check with the following configuration. Make sure your container answers the health checks as expected.

- Send HTTP GET request to `/` or a path that you define with the property `HealthCheckPath`.
- The timeout for the HTTP request is 10 seconds.
- An HTTP response is considered healthy if the status code is between 200 and 399.

Want to check the health of your containers? Use the following commands.

First, you need to fetch the ARN (Amazon Resource Name) of your target group. Replace `$STACK_NAME` with the name of your parent CloudFormation stack (e.g., `php-basic`).

```
aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`AppTargetGroupArn`].OutputValue' \
  --output text
```

The command returns the target group ARN, which should look similar to the following output:

```
arn:aws:elasticloadbalancing:[...]:targetgroup/php-[...]
```

Execute the following command to list the health status of all containers registered with the target group. Replace `$TARGET_GROUP_ARN` with the result from the previous command.

```
aws elbv2 describe-target-health \
  --target-group-arn $TARGET_GROUP_ARN
```

The output should look something like this, indicating one container passing and one container failing the health check:

```
{
  "TargetHealthDescriptions": [
    {
      "HealthCheckPort": "80",
      "Target": {
        "AvailabilityZone": "us-east-1b",
        "Id": "10.0.46.76",
        "Port": 80
      },
      "TargetHealth": {
        "State": "healthy"
      }
    },
    {
      "HealthCheckPort": "80",
      "Target": {
        "AvailabilityZone": "us-east-1a",
        "Id": "10.0.14.194",
        "Port": 80
      },
      "TargetHealth": {
        "State": "unhealthy",
        "Reason": "Target.Timeout",
        "Description": "Connection to target timed out"
      }
    }
  ]
}
```

Read the rest of this chapter to learn more about debugging a failed container.

Our examples (e.g., `php-basic`) include a working load balancer configuration. You will learn how to tailor the configuration of the ALB, listener, rule, and target group next.

In your temporary working environment, execute the following command to

change directory:

```
cd /src/php-basic/aws
```

Open the template `template.yml` of the parent stack. The following code excerpt shows the relevant modules:

```
Alb:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    # [...]
AlbListenerHttp:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    # [...]
AlbListenerHttps:
  Condition: HasCertificateArn
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    # [...]
AppTarget:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    # [...]
Redirect:
  Condition: HasCertificateArn
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    # [...]
```

The load balancer is configured to redirect HTTP requests to HTTPS. If you don't want to listen on port 80 at all, remove the following two modules: `AlbListenerHttp` and `Redirect`. Please note, to enable HTTPS you need to follow the instructions in section 3.4.

On top of that, each module comes with additional configuration options.

The alb module supports the following optional parameters:

- `BucketModule`: Capture access logs and store them in an S3 bucket by specifying the stack name of an s3-bucket module.
- `Scheme`: Either `internet-facing` (load balancer is accessible from the Internet) or `internal` (load balancer is only accessible from the private network or VPC).
- `IdleTimeoutInSeconds`: The number of seconds the load balancer waits

before closing an idle connection.

The following snippet shows the full-blown configuration of the `alb` module:

```
Alb:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      VpcModule: !GetAtt 'Vpc.Outputs.StackName'
      AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
      BucketModule: '' # optional
      Scheme: 'internet-facing' # optional
      IdleTimeoutInSeconds: '60' # optional
    TemplateURL: './node_modules/@cfn-modules/alb/module.yml'
```

The HTTP and HTTPS listener (`AlbListenerHttp` and `AlbListenerHttps`) do not offer any additional configuration options.

The ecs-alb-target module supports the following optional parameters:

- `DeregistrationDelayInSeconds`: How long does it take your application to answer a request in the worst case? This is the number of seconds to wait before removing a container from the load balancer during a deployment.

The following snippet shows the configuration of the `ecs-alb-target` module:

```
AppTarget:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      AlbModule: !GetAtt 'Alb.Outputs.StackName'
      AlbListenerModule: # [...]
      VpcModule: !GetAtt 'Vpc.Outputs.StackName'
      AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
      Priority: '2'
      HealthCheckPath: '/health-check.php'
    TemplateURL: '[...]'
```

We are not covering operating multiple applications behind the same load balancer in this book. If you are interested, check out the parameters of the ecs-alb-target module.

Managing and running your containers: ECS and Fargate

The Amazon Elastic Container Service (ECS) orchestrates containers. The service is fully managed by AWS and free of charge.

Historically, ECS started by adding a layer of abstraction on top of a fleet of EC2 instances (virtual machines). Distributing containers among a fleet of EC2 instances that you have to manage yourself is complicated. Therefore, we are not covering that approach at all.

Nowadays, AWS offers a fully-managed container infrastructure called AWS Fargate. No need to operate virtual machines any longer. Just tell ECS to launch containers on the Fargate compute engine.

When launching containers on Fargate, you allocate CPU and memory resources. AWS bills for each second a container is running. A few examples for containers running in US East (N. Virginia):

- Smallest possible configuration: a container with 0.25 vCPU and 0.5 GB memory costs about USD 9 per month.
- Medium-sized configuration: a container with 1 vCPU and 4 GB memory costs about USD 42 per month.
- Largest possible configuration: a container with 8 vCPU and 30 GB memory costs about USD 213 per month.

For details, see AWS Fargate Pricing.

Yes, we know that it seems to be cheaper to run containers on EC2 instances instead of Fargate. However, we highly recommend not doing so. Having to manage two layers (containers and EC2 instances) adds a lot of complexity and therefore hidden costs to your infrastructure. Not to mention that managing EC2 instances slows you down.

Next, you will learn about the basic components of ECS. A cluster defines the boundary for the compute infrastructure (EC2 or Fargate) and your containers. The following steps are needed to start a container:

1. Creating a **task definition**, including all the information needed to start a container. For example, the Docker image, as well as the required CPU and memory capabilities.
2. Creating a **task** based on the task definition. This will cause ECS to start a container.

In some cases, it is necessary to run multiple containers on the same machine. For example, we are launching an NGINX container as well as a PHP-FPM container on the same machine in our `php-basic` example application. In this scenario, low network latency between the two containers is key. Therefore, we are launching both containers on the same machine, allowing them to communicate without an additional network hop. Doing so is simple: add multiple containers to the task definition. ECS places all containers defined in the same task definition on the same machine.

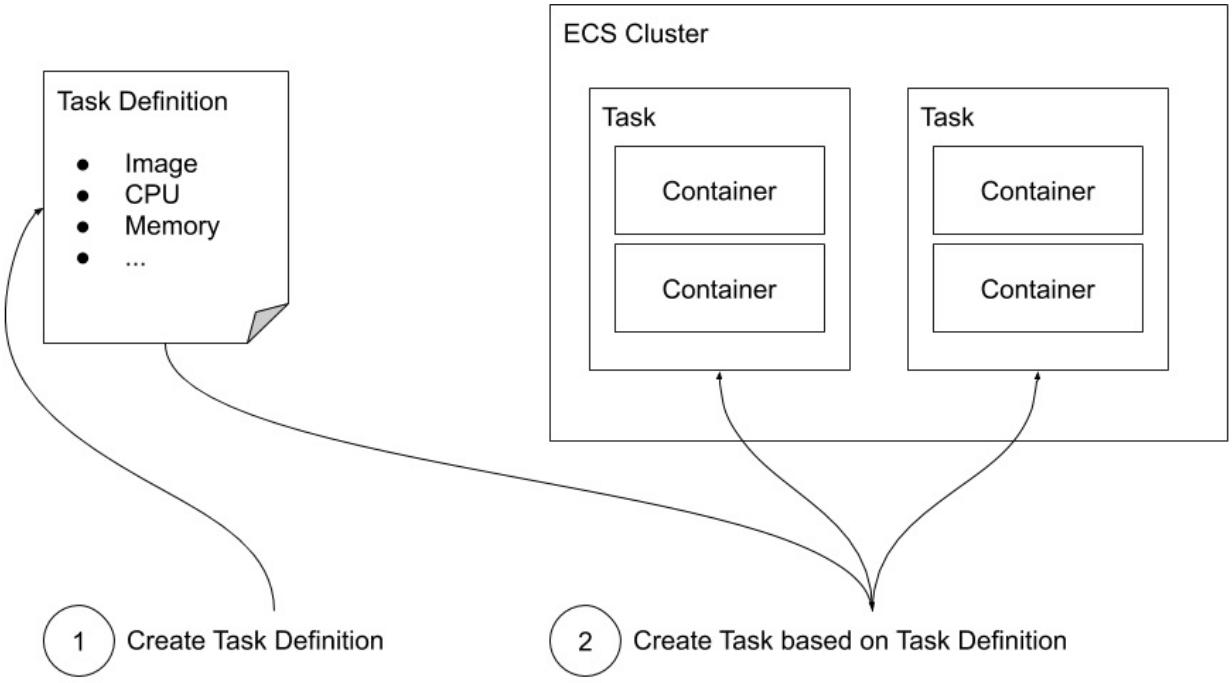


Figure 14: Create a task definition to be able to create a task which launches a container

There are two ways to create a task:

- **On-demand**: create a task for a finite job on-demand via the AWS Management Console or the AWS API. Typical use cases are batch processes, build jobs, etc.
- **Service**: make sure a specific number of tasks are running 24/7 to serve

requests. Typical use cases are HTTP servers, queue consumers, etc.

An ECS service makes sure the specified number of tasks, also called the **desired count**, is up and running. To do so, the ECS service will start or stop tasks as needed. As shown in the figure below, the ECS service fulfills the following functions:

- **High Availability**: spreads containers among different machines in different data centers (AZs), checks the health of the containers (e.g., by using the ALB health check), replaces failed containers.
- **Scalability**: starts and stops containers to reach the desired count. Auto-scaling can be used to increase or decrease the desired count based on the current load.
- **Service Discovery**: registers and deregisters containers with the load balancer.
- **Deployment**: executes rolling updates without service interruption to replace running containers with new containers based on a new image or configuration.

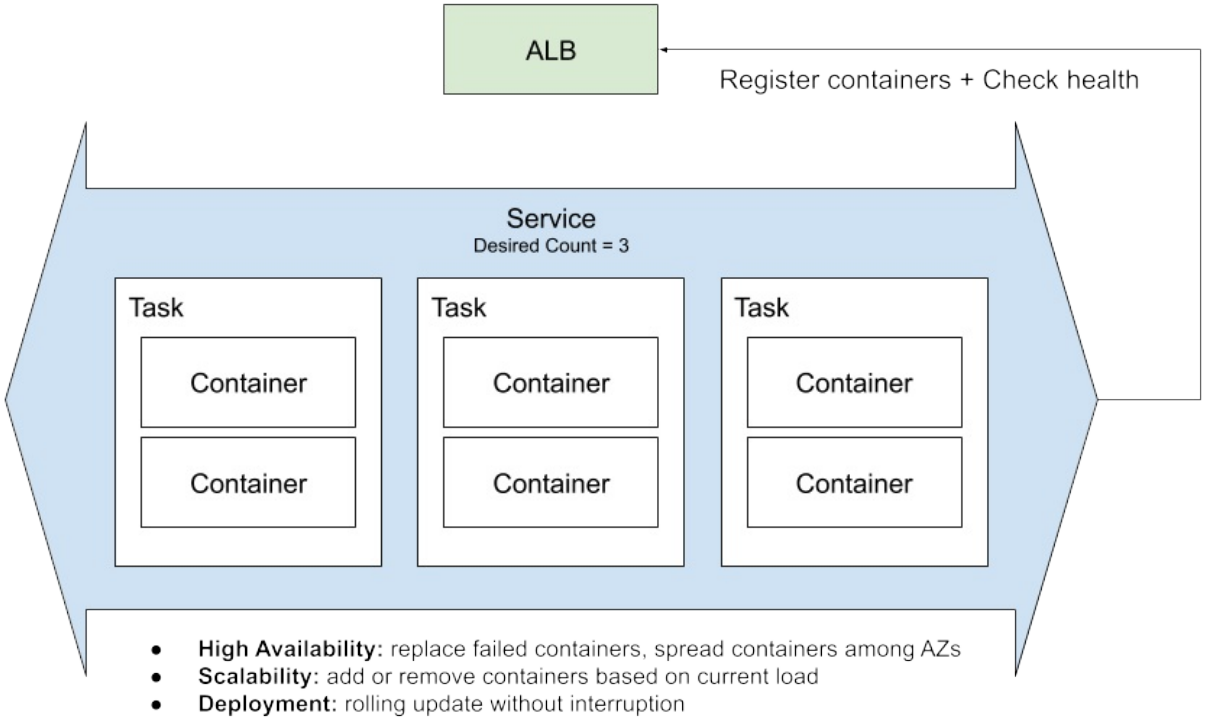


Figure 15: An ECS Service starts and stops tasks as needed

As this book is about deploying a web application — which includes an HTTP

server — we are making use of an ECS service.

Next, you will learn how to tailor the configuration to your specific needs.

In your temporary working environment, execute the following command to change directory:

```
cd /src/php-basic/aws
```

Within the `template.yml` file, you will find the configuration for the fargate-service module, as shown in the following snippet:

```
AppService:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      # [...]
      ProxyImage: !Ref ProxyImage
      ProxyPort: '80'
      AppImage: !Ref AppImage
      AppPort: '9000'
      AppEnvironment1Key: 'RDS_PASSWORD'
      AppEnvironment1SecretModule: # [...]
      AppEnvironment2Key: 'RDS_HOSTNAME'
      AppEnvironment2Value: # [...]
      Cpu: '0.25'
      Memory: '0.5'
      DesiredCount: '2'
      MaxCapacity: '4'
      MinCapacity: '2'
      LogsRetentionInDays: '14'
    TemplateURL: '[...]'
```

You might want to adjust the following parameters for your needs:

- `Cpu`: the number of virtual cores available to the containers of a task (allowed values: 0.25, 0.5, 1, 2, 4).
- `Memory`: the amount of memory available to the containers of a task (allowed values: 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30).
- `DesiredCount`: the initial number of tasks.
- `MaxCapacity`: the maximum number of tasks, the upper limit for auto-scaling.
- `MinCapacity`: the minimum number of tasks, the lower limit for auto-

scaling.

Keep in mind that increasing `Cpu`, `Memory`, `DesiredCount`, `MaxCapacity`, and `MinCapacity` will result in higher AWS costs.

Configuring a custom domain name and HTTPS: Route 53 and Certificate Manager

Up until now, your infrastructure has been missing two important features:

1. Your web application has not been accessible via a custom domain name. Instead, you have been using a domain name generated by AWS, similar to `demo-app[...].amazonaws.com`.
2. Connections from the browser to your web application have not been encrypted. The load balancer has only been listening for HTTP and not HTTPS.

Dependency This section requires you to buy or own a domain name (e.g., cloudonaut.io). The easiest way to follow this section is to register a domain name with Route 53. If you haven't done so, please follow the AWS documentation: Registering a New Domain. It is also possible to keep your domain at your current domain service. In that case, you have to adapt the instructions slightly where mentioned.

It's time to configure a custom domain name and HTTPS for your web application. The figure below shows the components involved:

- The *AWS Certificate Manager* is used to get a trusted SSL/TLS certificate for your domain name. A separate CNAME record is used to validate the certificate.
- A CNAME record is added to *Amazon Route 53* or any other domain name service for your domain name. The CNAME record points your domain name (e.g., `app.cloudonaut.io`) to the hostname of the ALB (e.g., `demo-app-xyz.us-east-1.elb.amazonaws.com`).

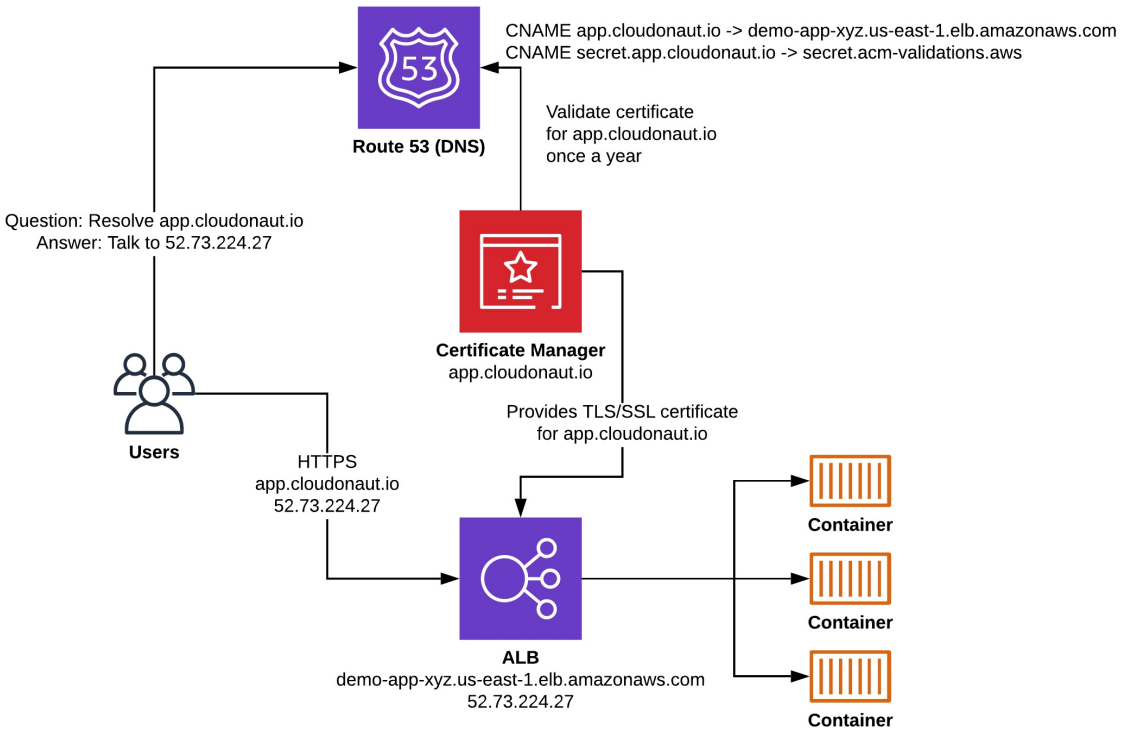


Figure 16: Using the Certificate Manager to generate SSL/TLS certificates and Route 53 to add DNS records

The first step is to create an SSL/TLS certificate for your domain name. Luckily, AWS provides public certificates free of charge.

Execute the following command to find out the ID of the hosted zone connected to your domain. Skip this step if you are not using Route 53.

```
aws route53 list-hosted-zones
```

The result should look similar to the following output.

```
{
  "HostedZones": [
    {
      "ResourceRecordSetCount": 2,
      "CallerReference": "BE50E7E5-AAF7-6897-8661-367FDF953881",
      "Config": {
        "PrivateZone": false
      },
      "Id": "/hostedzone/Z3L4U7A5QX3OGA",
      "Name": "cloudonaut.io."
    }
```

```
  ]
}
```

Find the hosted zone matching your domain and note its ID. In my example, the hosted zone ID is `Z3L4U7A5QX3OGA`.

Next, create a new certificate by executing the following command. Replace `$CUSTOM_DOMAN_NAME` with your domain name.

```
aws acm request-certificate --domain-name $CUSTOM_DOMAIN_NAME \
  --validation-method DNS
```

In my example, I'm using the domain name `app.cloudonaut.io`:

```
aws acm request-certificate --domain-name app.cloudonaut.io \
  --validation-method DNS
```

Note the ARN of the certificate shown in the output of the previous command:

```
{
  "CertificateArn": "arn:aws:acm:[...]:certificate/[...]a81289051"
}
```

In my example, it's:

```
arn:aws:acm:[...]:certificate/[...]a81289051
```

It is also possible to request a certificate for multiple domain names. Don't forget to replace `$CUSTOM_DOMAIN_NAME_*` with your domain names.

```
aws acm request-certificate --domain-name $CUSTOM_DOMAIN_NAME_1 \
  --validation-method DNS \
  --subject-alternative-name $CUSTOM_DOMAIN_NAME_2 \
$CUSTOM_DOMAIN_NAME_3
```

To validate your certificate, you need to create a CNAME record for each custom domain name included in the certificate.

The following command shows the information for the CNAME records you need to create for domain validation. Replace `$CERTIFICATE_ARN` with the ARN of the certificate you requested.

```
aws acm describe-certificate --certificate-arn $CERTIFICATE_ARN \
  --query 'Certificate.DomainValidationOptions[*].ResourceRecord' \
```

```
  --output text
```

In my example, the output looks as follows. Depending on whether you have created a certificate for one or multiple domain names, your output might have more than one line.

```
_4992593783e4347ca50333525d844a62.app.cloudonaut.io.
CNAME
_75df9596bad90f2ad47f058778c0b9ed.duyqrilejt.acm-validations.aws.
```

What does the output tell me? I need to create a CNAME record with name (`$NAME`) `_4992593783e4347ca50333525d844a62.app.cloudonaut.io.` and the value (`$VALUE`) `_75df9596bad90f2ad47f058778c0b9ed.duyqrilejt.acm-validations.aws.`.

Of course, the name and value will differ for you.

So how do you create a CNAME record? If you are not using Route 53 to manage your domain, please consult the documentation of your domain name service provider. When using Route 53, follow the next steps.

The following command creates a CNAME record that is used by the certificate manager to validate your domain name. The `--change-batch` parameter value is complex, therefore, you offload it into a file. Create the file `change-batch.json` with the following content in your working directory with your prefered editor (e.g., `nano change-batch.json`). Replace `$NAME` and `$VALUE` with the outputs from the previous step.

```
{
  "Changes": [{
    "Action": "UPSERT",
    "ResourceRecordSet": {
      "Name": "$NAME",
      "Type": "CNAME",
      "TTL": 60,
      "ResourceRecords": [{
        "Value": "$VALUE"
      }]
    }
  }]
}
```

In my case, that results in the following file:

```
{
  "Changes": [{
    "Action": "UPSERT",
    "ResourceRecordSet": {
      "Name": "_4992593783e[...]33525d844a62.app.cloudonaut.io.",
      "Type": "CNAME",
      "TTL": 60,
      "ResourceRecords": [{
        "Value": "_75d[...]0bed.duyqrilejt.acm-validations.aws."
      }]
    }
  }]
}
```

And run the command. Replace `$HOSTED_ZONE_ID` with your hosted zone ID.

```
aws route53 change-resource-record-sets \
  --hosted-zone-id $HOSTED_ZONE_ID \
  --change-batch 'file://change-batch.json'
```

In my case, that results in the following command:

```
aws route53 change-resource-record-sets \
  --hosted-zone-id Z3L4U7A5QX3OGA \
  --change-batch 'file://change-batch.json'
```

You need to repeat this command for each custom domain name you added to your certificate request.

It's time to grab a coffee or tea now. It will take up to 15 minutes until the certificate is validated. Use the following command to wait for the certificate validation. Replace `$CERTIFICATE_ARN` with the ARN of the certificate you requested.

```
aws acm wait certificate-validated \
  --certificate-arn $CERTIFICATE_ARN
```

The command will exit as soon as your certificate is ready to use.

There is one step missing. You need to re-deploy your cloud infrastructure to enable HTTPS with the certificate you created.

Make sure you have started your working environment to prepare the deployment:

```
cd /src/php-basic/aws
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

Deploy or redeploy your infrastructure. Replace `$STACK_NAME` with the name of your parent CloudFormation stack (e.g., `php-basic`) and `$CERTIFICATE_ARN` with your newly created certificate.

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name $STACK_NAME --capabilities CAPABILITY_IAM \
  --parameter-overrides CertificateArn=$CERTIFICATE_ARN
```

The load balancer listens on port 443 for HTTPS connections now. When a user accesses your web application via HTTP, the load balancer will redirect to HTTPS immediately.

There is only one step missing to connect your domain with the load balancer: creating a CNAME pointing from your domain name to the load balancer.

Execute the following command to fetch the DNS name of the load balancer. Replace `$STACK_NAME` with the name of your parent CloudFormation stack (e.g., `php-basic`).

```
aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`AlbDnsName`].OutputValue' \
  --output text
```

For me, the command returns the following output:

```
php-LoadB-1BCK08OLNTQR5-77309766.us-east-1.elb.amazonaws.com
```

In my case, I want to create a CNAME pointing from `app.cloudonaut.io` to `php-LoadB-1BCK08OLNTQR5-77309766.us-east-1.elb.amazonaws.com`.

Use the following command to create a CNAME record with Route 53. If you are not using Route 53 to manage your domain, please consult the documentation of your domain name service provider.

Re-use the `change-batch.json` file with the following content in your working directory with your prefered editor (e.g., `nano change-batch.json`). Replace

`$CUSTOM_DOMAIN` and `$ALB_DNS_NAME` with the outputs from the previous step.

```
{
  "Changes": [{
    "Action": "UPSERT",
    "ResourceRecordSet": {
      "Name": "$CUSTOM_DOMAIN",
      "Type": "CNAME",
      "TTL": 60,
      "ResourceRecords": [{
        "Value": "$ALB_DNS_NAME"
      }]
    }
  }]
}
```

In my case, that results in the following file:

```
{
  "Changes": [{
    "Action": "UPSERT",
    "ResourceRecordSet": {
      "Name": "app.cloudonaut.io",
      "Type": "CNAME",
      "TTL": 60,
      "ResourceRecords": [{
        "Value":"php-LoadB-1BC[...]766.us-east-1.elb.amazonaws.com"
      }]
    }
  }]
}
```

And run the command. Replace `$HOSTED_ZONE_ID` with your hosted zone ID.

```
aws route53 change-resource-record-sets \
  --hosted-zone-id $HOSTED_ZONE_ID \
  --change-batch 'file://change-batch.json'
```

In my case, that results in the following command:

```
aws route53 change-resource-record-sets \
  --hosted-zone-id Z18W2IF733UZVC \
  --change-batch 'file://change-batch.json'
```

That's it. Point your browser to your custom domain name. Your web application will show up.

Warning It might take minutes or even hours until the custom domain name works as expected depending on the settings of your hosted zone and the DNS servers used.

Storing and querying your data: RDS Aurora Serverless

The database is a central component of the architecture. The containers are stateless. Therefore, all data needs to be stored in the database. We are using RDS Aurora Serverless (MySQL compatible edition), a database service with two important features:

- Scalable storage capacity distributed among multiple data centers (up to 64 TiB).
- Scalable compute capacity with the ability to scale vertically from 1 to 256 ACU (1 ACU corresponds to approximately 2 GB of memory and 1 vCPU).

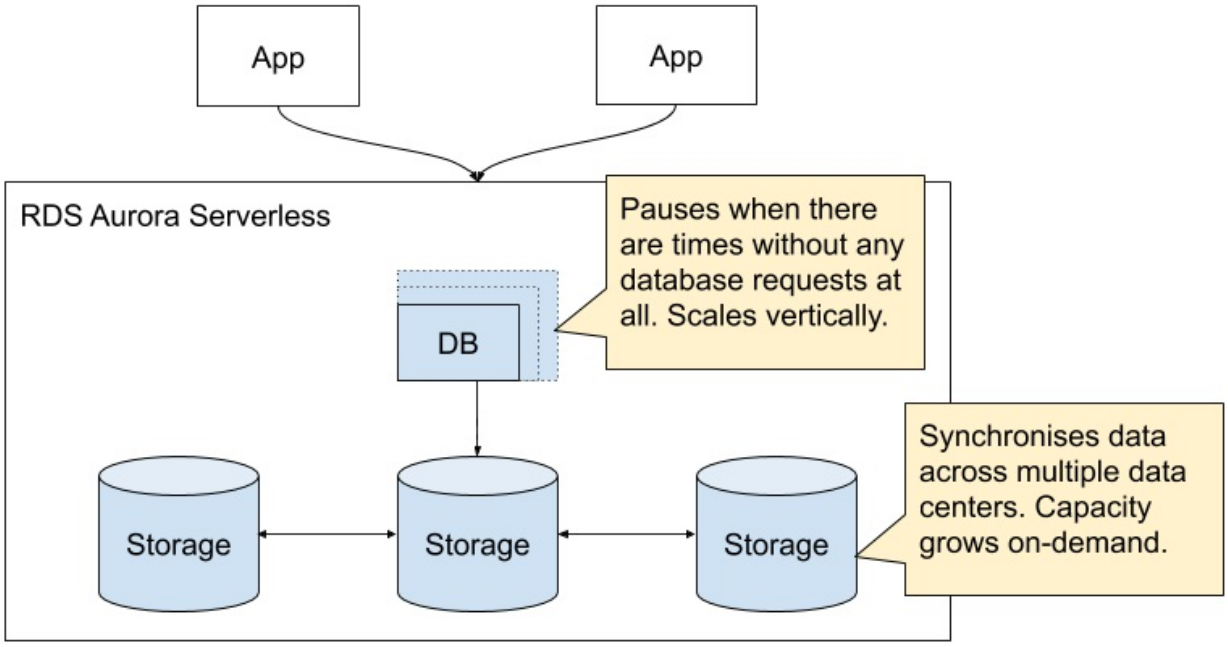


Figure 17: RDS Aurora Serverless provides scalable storage and compute capacity

With RDS Aurora Serverless, the storage and compute capacity scale automatically, which minimizes database maintenance. Luckily, RDS Aurora Serverless works without much configuration. There are only a few parameters to tweak.

In your temporary working environment, execute the following command to change directory:

```
cd /src/php-basic/aws
```

Within the `template.yml` file, you will find the configuration for the rds-aurora-serverless module, as shown in the following snippet:

```
AuroraServerlessCluster:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      VpcModule: # [...]
      ClientSgModule: # [...]
      KmsKeyModule: !GetAtt 'Key.Outputs.StackName'
      AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
      SecretModule: !GetAtt 'Secret.Outputs.StackName'
      DBName: test
      DBMasterUsername: master
      AutoPause: 'true'
      SecondsUntilAutoPause: '900'
      MinCapacity: '1'
      MaxCapacity: '2'
      EngineVersion: '5.6.10a'
    TemplateURL: '[...]'
```

You might want to adapt the following parameters to your needs:

- `DBName`: the name of the initial database created during the infrastructure setup.
- `AutoPause`: Should the database be paused at all? Set to `false` to disable pausing.
- `SecondsUntilAutoPause`: the number of seconds after inactivity until the database is paused (allowed values: 1-86400).
- `MinCapacity`: the minimum compute capacity (CPU and memory) of the database cluster (allowed values: 1, 2, 4, 8, 16, 32, 64, 128, 256).
- `MaxCapacity`: the maximum compute capacity (CPU and memory) of the database cluster (allowed values: 1, 2, 4, 8, 16, 32, 64, 128, 256).

Warning Keep in mind that increasing `MinCapacity` or `MaxCapacity` will result in higher AWS costs.

For additional parameters, check out the module's documentation cfn-

modules/rds-aurora-serverless on GitHub.

How to administer your database? For example, to execute SQL statements or to import data? The database itself is not reachable from the Internet. Instead, we recommend using a web-based tool called phpMyAdmin.

> **Dependency** To deploy phpMyAdmin, you need to finish section 3.4, where you create a CNAME record pointing to your load balancer's DNS name and an SSL/TLS certificate.

Execute the following command to retrieve your certificates from Amazon Certificate Manager (ACM).

```
aws acm list-certificates
```

The command returns a list of certificates:

```
{
  "CertificateSummaryList": [
    {
      "CertificateArn": "arn:aws:acm:[...]:certificate/[...]1e4de",
      "DomainName": "app.cloudonaut.io"
    }
  ]
}
```

Note the `CertificateArn` from the certificate that matches the custom domain name you want to use to access phpMyAdmin. I'm noting the ARN `arn:aws:acm:us-east-1:111111111111:certificate/4c37657c-a94c-4b5e-a101-2474d4b1e4de` as I want to use the domain `app.cloudonaut.io` to access phpMyAdmin.

Next, you need to deploy your parent stack. The following example shows the required steps for the php-basic example. The same commands work with our example applications written in other programming languages (like Ruby, Python, and Java) as well.

Replace the following placeholders:

- `$STACK_NAME` with the name of your parent CloudFormation stack (e.g., `php-basic`)
- `$ADMIN_EMAIL` with your email address

- `$CERTIFICATE_ARN` with the ARN of the certificate from the previous step
- `$DEFAULT_DOMAIN` with the custom domain name you want to use to access phpMyAdmin

```
cd /src/php-basic/aws
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
aws cloudformation deploy --template-file .template.yml \
  --stack-name $STACK_NAME --capabilities CAPABILITY_IAM \
  --parameter-overrides AdminEmail=$ADMIN_EMAIL \
CertificateArn=$CERTIFICATE_ARN \
PhpMyAdminDomain=$DEFAULT_DOMAIN \
EnablePhpMyAdmin=true
```

You will receive two emails:

1. Subscription Confirmation: You will use this in 3.6.
2. Temporary Password: Email with the initial password to authenticate with phpMyAdmin required in a few steps.

Next, use the following commands to retrieve the URL for phpMyAdmin as well as the ARN of the secret containing the password of the database user `master`. Replace `$STACK_NAME` with the name of the parent stack (e.g., `php-basic`).

```
aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`PhpMyAdminUrl`].OutputValue' \
  --output text

aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`DatabaseSecretArn`].OutputValue' \
  --output text
```

Use the following command to retrieve the password for the database user master. Replace `$SECRET_ARN` with the result from the previous command.

```
aws secretsmanager get-secret-value --secret-id $SECRET_ARN \
  --query SecretString --output text
```

Point your web browser to the phpMyAdmin URL. There are two authentication steps:

1. Authenticate with your email address (see parameter `AdminEmail`). You received the initial password via email.
2. Authenticate with the database user `master` and the password you retrieved from Secrets Manager.

The following screenshot shows phpMyAdmin browsing the table `site_hits`, which is part of the database `test`:

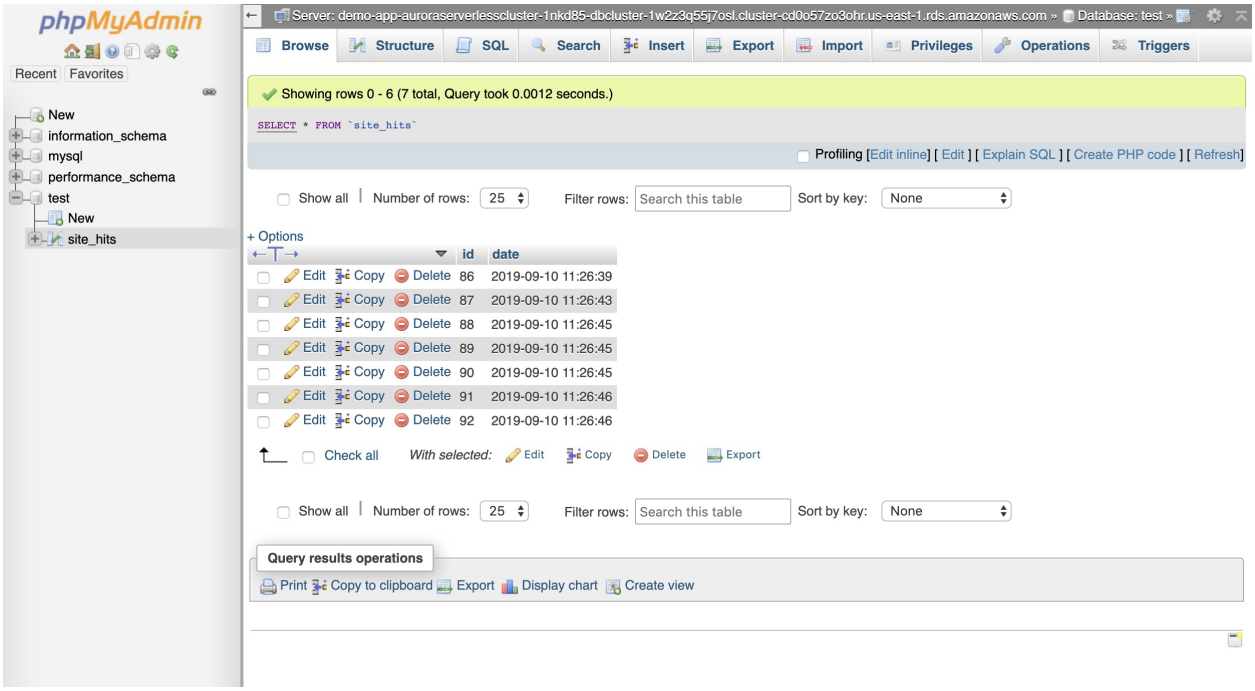


Figure 18: Browsing your database with phpMyAdmin

Use phpMyAdmin to modify your databases, tables, and data. You can also import data from your existing database or export all your data. Consult the phpMyAdmin documentation for more details.

Monitoring and debugging: CloudWatch

We promised to deliver a production-ready infrastructure for your web application. Therefore, our examples include alarms that monitor metrics, and logging by default. The following figure shows that all parts of the infrastructure are sending metrics to a service called Amazon CloudWatch. The metrics include CPU utilization of your containers, the number of database connections, the number of HTTP 5XX errors caused by the load balancer, and many more. Alarms monitor relevant metrics and send notifications to SNS (Amazon Simple Notification Service). SNS delivers the alarm notifications to you via email or other destinations.

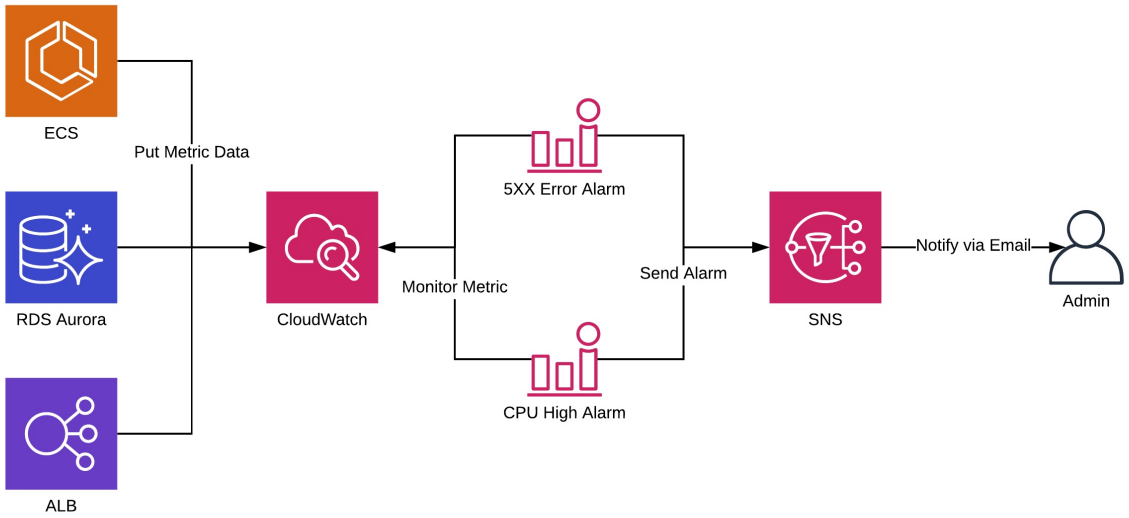


Figure 19: Monitoring with CloudWatch: metrics and alarms

The following alarms make sure you are the first to know when your web application is not working as expected:

HTTPCodeELB5XXTooHighAlarm: An HTTP request was answered with a 5XX status code (server-side error) by the load balancer, most likely because there was no healthy container running.

Check if there are healthy targets registered with the target group (see 3.2) or check NGINX and application logs for error messages (see 3.5).

TargetConnectionErrorCountTooHighAlarm: The load balancer was not able to establish a connection to one of the containers.

Check if there are healthy targets registered with the target group (see 3.2) or check NGINX and application logs for error messages (see 3.5).

HTTPCodeTarget5XXTooHighAlarm: Your application responded with a 5XX status code (server-side error).

Check NGINX and application logs for error messages (see 3.5).

RejectedConnectionCountTooHighAlarm: A client connection to the load balancer was rejected.

Wait up to 60 minutes. If the problem persists, open an AWS support ticket.

CPUUtilizationTooHighAlarm: The CPU utilization of your containers is too high.

Check NGINX and application logs for error messages (see 3.5) or increase the maximum number of containers or CPU capacity (see 3.3).

You will learn how to make sure alarms are reaching you via email next.

In your temporary working environment, execute the following command to change directory:

```
cd /src/php-basic/aws
```

Within the `template.yml` file, you will find the configuration for the alerting module, as shown in the following snippet:

```
Alerting:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      Email: !Ref AdminEmail
      # HttpsEndpoint: 'https://api.marbot.io/v1/endpoint/xyz'
    TemplateURL: './node_modules/@cfn-modules/alerting/module.yml'
```

You might want to adapt the following parameters to your needs:

- `Email`: the email address that should receive alarm notifications. Use the parameter `AdminEmail` to reference your email address as described in 3.5.
- `HttpsEndpoint`: the HTTPS endpoint that should receive alarm notifications. It is used to integrate with incident management tools like our Slack bot marbot.io.

Our goal is to send alarm notifications only when the user experience is affected. However, you might want to get an overview of the state and utilization of your infrastructure from time to time. Use the predefined CloudWatch dashboard as illustrated in the screenshot below to do so. The dashboard includes:

- **ALB Errors**: the 4XX (client-side) and 5XX (server-side) error rate of your web application.
- **ALB Requests + Latency**: the number of requests, as well as the latency (99 and 95 percentiles).
- **ECS Utilization**: the CPU and memory utilization of your containers.

- **RDS IOPS + Capacity**: the capacity of your database, as well as the I/O throughput (IOPS).
- **RDS Latency**: the latency of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
- **RDS Queries**: the number of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.

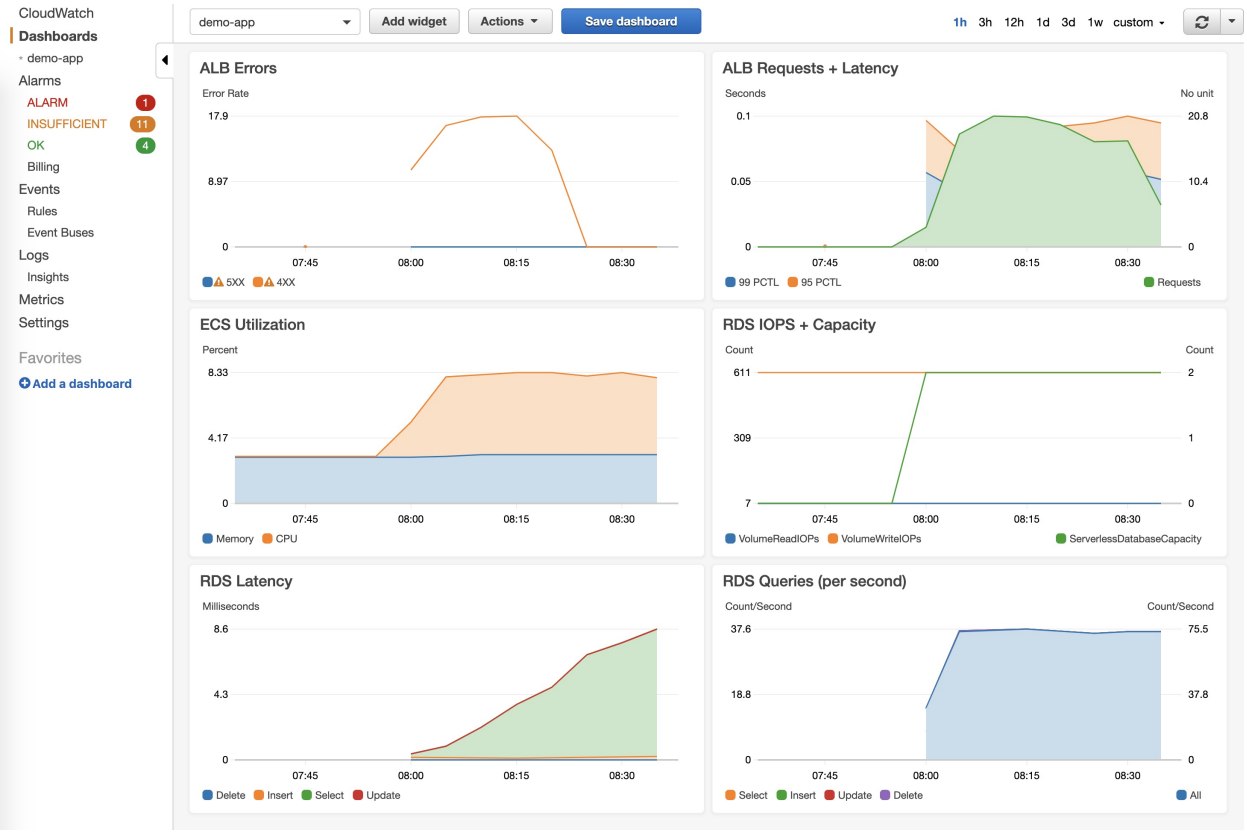


Figure 20: Screenshot of the CloudWatch dashboard monitoring the whole infrastructure

Execute the following command from your working environment to fetch the dashboard's URL. Replace `$STACK_NAME` with the name of the parent stack (e.g., `php-basic`).

```
aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`DashboardUrl`].OutputValue' \
  --output text
```

In summary, metrics and alarms allow you to monitor your cloud infrastructure and web application. One important piece for debugging is missing: log messages. As shown in the following figure, whenever your application writes a message to standard output (stdout) and standard error (stderr), these messages are automatically pushed to a log group in CloudWatch Logs. The log group collects all the log messages and stores them for 14 days. On top of that, you can search and analyze the log messages for debugging purposes whenever needed.

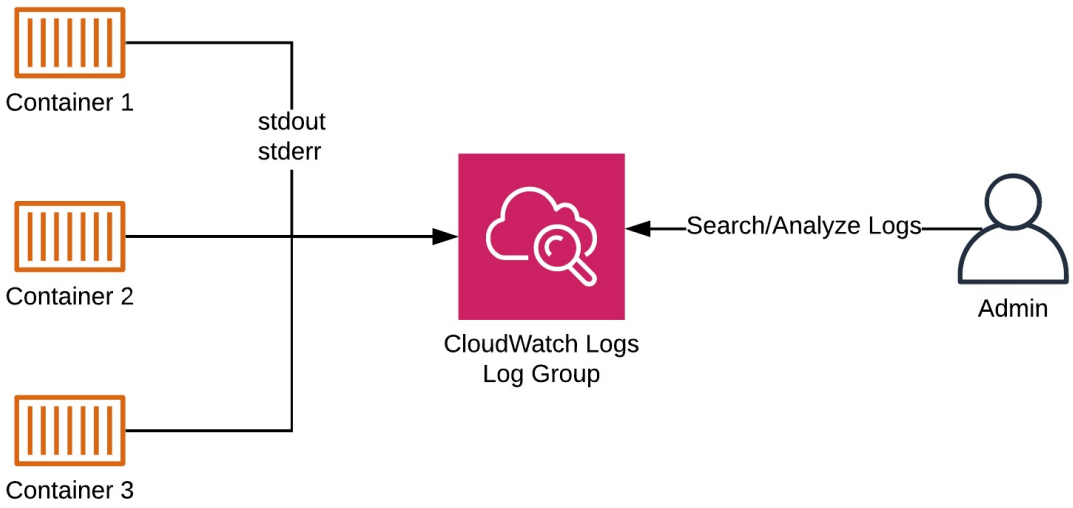


Figure 21: Centralized log management with CloudWach Logs

Execute the following command from your working environment to fetch the URL pointing to CloudWatch Logs Insights. Replace `$STACK_NAME` with the name of the parent stack (e.g., `php-basic`).

```
aws cloudformation describe-stacks --stack-name $STACK_NAME \
  --query \
'Stacks[0].Outputs[?OutputKey==`AppLogsUrl`].OutputValue' \
  --output text
```

The following screenshot shows CloudWatch Logs Insights in action:

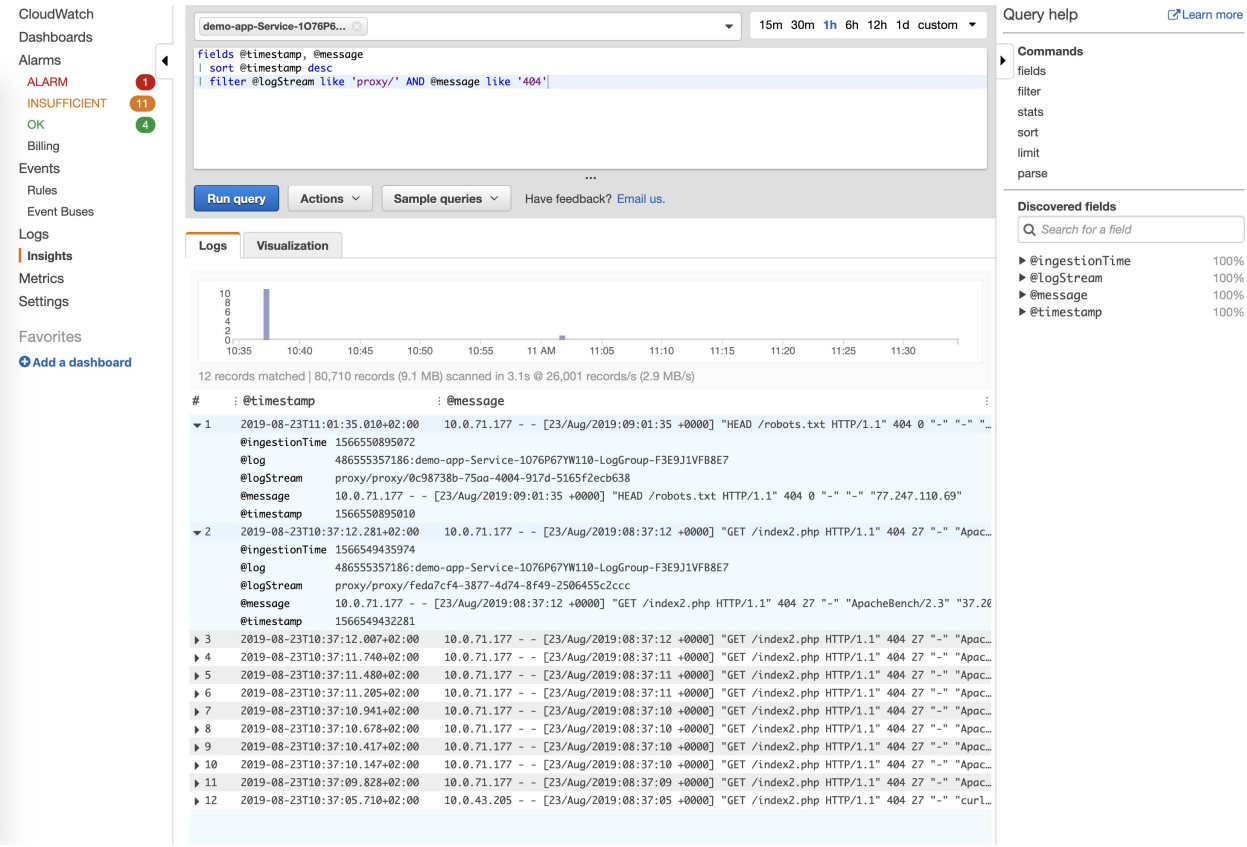


Figure 22: Screenshot CloudWatch Logs in Action

To search through your logs, you need to:

1. Modify the query.
2. Select a time span.
3. Hit the **Run query** button.

> **Warning** Choose a time span as short as possible to avoid unnecessary costs when searching the log messages.

Let's start with a simple query filtering all log messages from the proxy container (NGINX):

```
fields @timestamp, @message
| sort @timestamp desc
| filter @logStream like 'proxy/'
```

And a similar query to filter all log messages from the app container (e.g., PHP-FPM):

```
fields @timestamp, @message
| sort @timestamp desc
| filter @logStream like 'app/'
```

This more advanced query filters all log messages from the proxy container (e.g., NGINX) containing the search term “404” in the log message:

```
fields @timestamp, @message
| sort @timestamp desc
| filter @logStream like 'proxy/' AND @message like '404'
```

Another example: this query filters all log messages from the app container (e.g., PHP-FPM) containing the search term “error” in the log message:

```
fields @timestamp, @message
| sort @timestamp desc
| filter @logStream like 'app/' AND @message like 'error'
```

Want to learn more about the query syntax? Check out the CloudWatch Logs Insights Query Syntax.

In summary, with CloudWatch metrics, alarms, and logs, monitoring and debugging your web application is simple.

Running scheduled jobs (cron) in the background

Many web applications use scheduled jobs to automate recurring tasks, such as:

- Generating and sending a monthly report.
- Disabling users who haven't logged in for more than 365 days.
- Deleting stale data from the database.

Most likely, you have used cron — a time-based job scheduler to trigger scheduled jobs. However, using a second process to run cron jobs in a container does not comply with Docker best practices. Remember the best practices when working with Docker containers from chapter 2?

Use one process per container. If your application consists of more than one process, split them up into multiple containers. For example, if you run NGINX and PHP-FPM, create two containers.

It is not very cost-effective to run a container on Fargate 24/7 to execute a job a few times per day.

Luckily, there is an alternative to cron when working with containers on AWS. As shown in the figure below, three components work together to schedule jobs:

- **CloudWatch Events Rule**: triggers the state machine based on a schedule.
- **Step Functions**: a state machine orchestrating simple or complex workflows. In this scenario, the state machine starts a container and waits until the container exits.
- **Fargate**: the computing engine for the container executing the scheduled job.

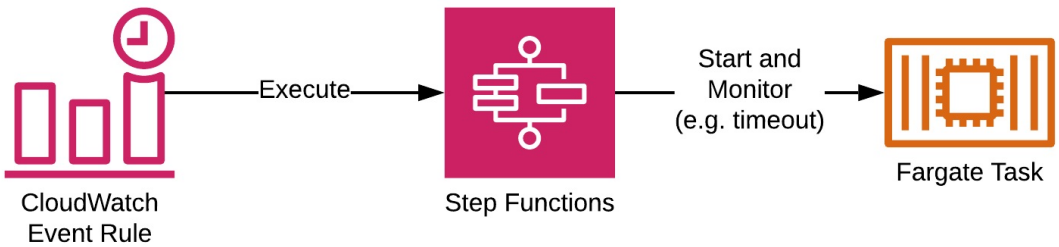


Figure 23: The CloudWatch Events Rule triggers a Step Function which starts a Fargate task

Our application includes a scheduled job. It resets the site hits counter in the database every 5 minutes. However, the scheduled task is disabled by default. Update your parent stack with the following commands to enable the scheduled task. Replace `$STACK_NAME` with the name of the parent stack (e.g., `php-basic`).

```
cd /src/php-basic/aws
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
aws cloudformation deploy --template-file .template.yml \
  --stack-name $STACK_NAME --capabilities CAPABILITY_IAM \
  --parameter-overrides EnableScheduledTask=true
```

The scheduled task will clear the site hits stored in the database every 5 minutes from now on.

Within the `template.yml` file, you will find the configuration for the [fargate-scheduled-task] module, as shown in the following snippet:

```
ScheduledTask:
  Type: 'AWS::CloudFormation::Stack'
  Properties:
    Parameters:
      VpcModule: !GetAtt 'Vpc.Outputs.StackName'
      ClusterModule: !GetAtt 'Cluster.Outputs.StackName'
      AlertingModule: !GetAtt 'Alerting.Outputs.StackName'
      ClientSgModule1: !GetAtt 'AuroraServerlessClientSg.[..]'
      AppImage: !Ref ScheduledTaskImage
      AppEnvironment1Key: 'DATABASE_PASSWORD'
      AppEnvironment1SecretModule: !GetAtt 'Secret.Outputs.StackName
      AppEnvironment2Key: 'DATABASE_HOST'
      AppEnvironment2Value: !GetAtt 'AuroraServerlessCluster.[..]'
      AppEnvironment3Key: 'DATABASE_NAME'
      AppEnvironment3Value: 'test'
      AppEnvironment4Key: 'DATABASE_USER'
      AppEnvironment4Value: 'master'
      Cpu: '0.25'
      Memory: '0.5'
      LogsRetentionInDays: '14'
      ScheduleExpression: 'rate(5 minutes)'
      Timeout: '300'
    TemplateURL: '[...]'
```

Use the following parameters to configure the recurrence and timeout of the scheduled job:

- `ScheduleExpression`: a schedule expression defined in cron or rate style (see examples below).
- `Timeout`: terminate the container executing the scheduled job after X seconds.

A few examples for schedule expressions in cron style:

Schedule Expression	Explanation
`cron(0 6 * * ? *)`	06:00 am (UTC) every day
`cron(0 12 * * MON-FRI *)`	12:00 am (UTC) from Monday to Friday
`cron(0 8 1 * ? *)`	08:00 am (UTC) every first day of the month

And more examples for schedule expressions in rate style:

Schedule Expression	Explanation
`rate(15 minutes)`	Every 15 minutes
`rate(1 hour)`	Every hour
`rate(14 days)`	Every 14 days

See Schedule Expressions for Rules for more detailed explanations.

On top of that, use the following parameters to configure the container that is executing the scheduled job:

- `AppImage`: the name and tag of the Docker image, that is used to launch a container to execute your scheduled job. We are using a Docker image including a mysql client within our example. Instead, you could also use the same image that includes your application and run some scripts that are bundled together with your application (e.g., PHP scripts).
- `AppCommand`: optionally, override the default command (defined by `CMD` in your Dockerfile) that is executed when the container starts.
- `AppEnvironment1Key`: the key of environment variable #1.
- `AppEnvironment1Value`: the value of environment variable #1.
- `AppEnvironment1SecretModule`: use the secret module to insert an auto-generated password (e.g., database password) instead of a value.

- `Cpu`: the number of virtual cores available to the containers of the task (allowed values: 0.25, 0.5, 1, 2, 4).
- `Memory`: the amount of memory available to the containers of the task (allowed values: 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30).

Warning Keep in mind that increasing `Cpu` and `Memory` will result in higher AWS costs.

In case you are not planning to continue with the next chapter in a timely manner, please delete the AWS infrastructure and repositories to avoid unnecessary costs. Otherwise, keep the infrastructure up and running.

```
cd /src/php-basic/aws
bash cleanup.sh
```

You have learned about the building blocks of the Rapid Docker on AWS architecture in this chapter: ALB, ECS, Fargate, and RDS Aurora Serverless. On top of that, this chapter covered how to configure a custom domain name, monitor your application, and run scheduled jobs. You will learn how to automate the process of making source code changes and deploying to production in the following chapter.

Deploying your source code and infrastructure continuously

So far, you have worked with files on your computer and have used the AWS CLI to interact with AWS. Two problems arise when you work in a team:

1. It's hard to share the files with a colleague, and it gets chaotic if both of you make changes to the files.
2. You are the only one on your team who knows how to deploy the application to AWS. You have to build the Docker images, push them, and update the CloudFormation stack. A lot of knowledge is needed to perform the steps in the right order. On top of that, the necessary tools are only installed on your machine.

In this chapter, you will learn to solve both challenges using a version control system and a deployment pipeline.

Versioning your source code with Git: AWS CodeCommit

A Version Control System (VCS) manages changes to documents like source code. Each change results in a new revision identified by a number. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and merged. Git is a popular VCS (other implementations exist, such as: SVN, Mercurial, CVS, and many more).

Benefits of version control

Version control empowers a team to work together on the same files. Version control also captures the complete history of all documents, which makes it possible to travel back in time to understand how a document evolved. Last but not least, with version control, you can undo any change by restoring a previous revision.

Introducing Git

You will learn next how to use Git to manage the source code of your application and infrastructure.

> **Dependency** Before you proceed, please make sure you have followed "Installing Docker" and "Configuring your temporary working environment" as described in chapter 1.

A short reminder of how to start your temporary working environment:

On Mac and Linux

Open a Terminal and change into the source code directory.

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment.

```
bash start.sh
```

On Windows

Open PowerShell and change into the book's source code directory:

```
cd docker-on-aws-code
```

And run the following command to start your temporary working environment.

```
powershell -ExecutionPolicy ByPass -File start.ps1
```

Now that you have started the working environment on your local machine (MacOS, Linux, or Windows), go to the directory of the example from chapter 2. You will use this example for your Git experiments:

```
cd /src/php-basic
```

The first step is to create a Git repository. The following command turns a directory into a repository:

```
git init
```

The next command shows the status of your repository:

```
git status -uall
```

The output is quite verbose. Let's examine each chunk:

```
On branch master
```

By default, Git creates a master branch that you will use in this book. We are not covering working with multiple branches in this book. If you are interested, check out the Pro Git book chapter about branches.

```
No commits yet
```

A commit creates a new revision, it contains a bunch of changes to files, a timestamp, the person who made the change, and a short commit message. Last but not least, you see all the files with changes in your repository.

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
  aws/package-lock.json
  aws/package.json
  aws/template.yaml
  [...]
  img/it-works.gif
  index.php
  lib/helper.php

nothing added to commit but untracked files present
(use "git add" to track)
```

You can now select the files that should be included in your first commit. To do so, you add the files to the so called staging area. git add path/to/file adds a single file, git add -A adds all files with changes to the staging area. Add all files to the staging area:

```
git add -A
```

See how the output of git status changes after adding the files:

```
git status -uall
```

You will now see changes to be committed:

```
[...]
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   aws/package-lock.json
  new file:   aws/package.json
  new file:   aws/template.yaml
  [...]
  new file:   img/it-works.gif
  new file:   index.php
  new file:   lib/helper.php
```

If you are happy with the files in the staging area, it's time to commit them. But before that, you have to configure Git slightly with an editor. Replace the dummy data with your real email and name.

```
nano /root/.gitconfig
```

Ensure that the following section exists:

```
[user]
    name = Your Name
```

```
    email = you@example.com
```

Hit CTRL + X to save, and confirm by pressing y and ENTER. Finally, you can commit your changes:

```
git commit -m 'initial commit'
```

Once again, see how the output of git status changes after adding the files:

```
git status -uall
```

No changes are visible anymore:

```
On branch master
nothing to commit, working tree clean
```

If you now make a change to a file, the game continues.

So far, the Git repository is stored on your local machine. If you want to work together with others or build a deployment pipeline, they need access to your machine. Granting them access to your machine is not a good solution because your machine is likely not available 24/7. It's easier to use a remote machine to do that. Thanks to Git, it is easy to synchronize a local repository with a remote repository.

Using a remote Git repository: CodeCommit

AWS offers remote Git repositories through AWS CodeCommit. CodeCommit is a fully managed service that you can use to store your Git repositories safely in AWS. GitHub, Bitbucket, and GitLab offer remote Git repositories as well but lack strong integration with AWS (e.g., IAM to manage permissions). You will now learn to use CodeCommit as a remote Git repository.

First, you have to create the CodeCommit repository. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name php-basic
```

The output will be similar to the JSON below. You will need the value of the cloneUrlHttp attribute later.

```
{
```

```
  "repositoryMetadata": {
    "repositoryName": "php-basic",
    "cloneUrlHttp": "https://git-codecommit[...]/php-basic",
    [...]
  }
}
```

Inside the `php-basic` directory, add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/php-basic
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/php-basic
```

It's time to sync your repository with the remote repository. Uploading your changes in Git is called pushing (to push changes) as shown by the next command:

```
git push -u origin master
```

The following output indicates that the changes are now pushed.

```
Counting objects: 28, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (28/28), 1.86 MiB | 450.00 KiB/s, done.
Total 28 (delta 1), reused 0 (delta 0)
[...]
```

Syncing changes from the remote to your machine is called pulling (to pull changes) and can be performed with the following command:

```
git pull
```

In this case, the output tells us that no new changes are available:

```
Current branch master is up to date.
```

The following figure demonstrates the use of a remote Git repository by a team:

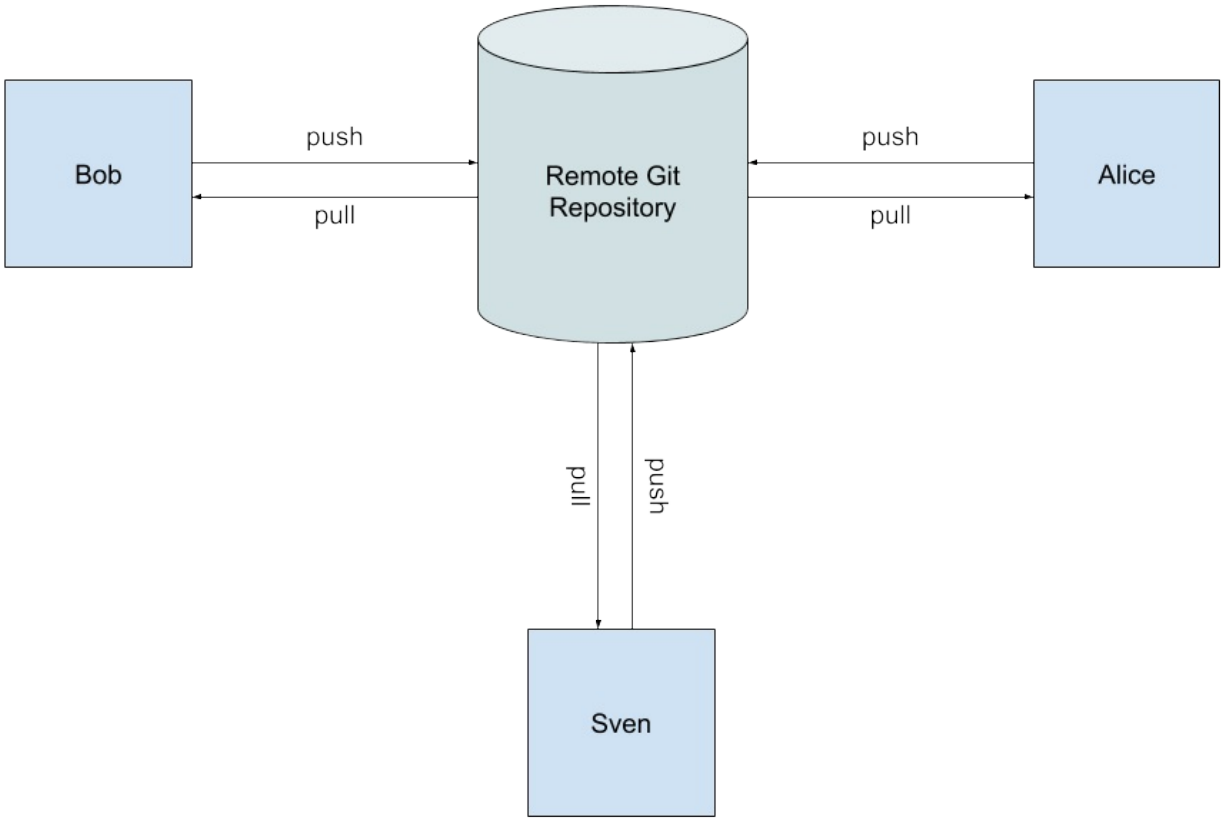


Figure 24: A team using a remote Git repository

That's all you need to know for now about Git. If you want to learn more, check out the free Pro Git book. One last thing about CodeCommit: You can also view and edit files online in the CodeCommit Management Console (you might need to set the AWS region at the very top right). Click on your repository to explore the content online.

Setting up a deployment pipeline: AWS CodeBuild

“A deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users” (Continuous Delivery, Humble/Farley). Every change to your code (e.g., a `git push`) triggers the pipeline as shown in the next figure.

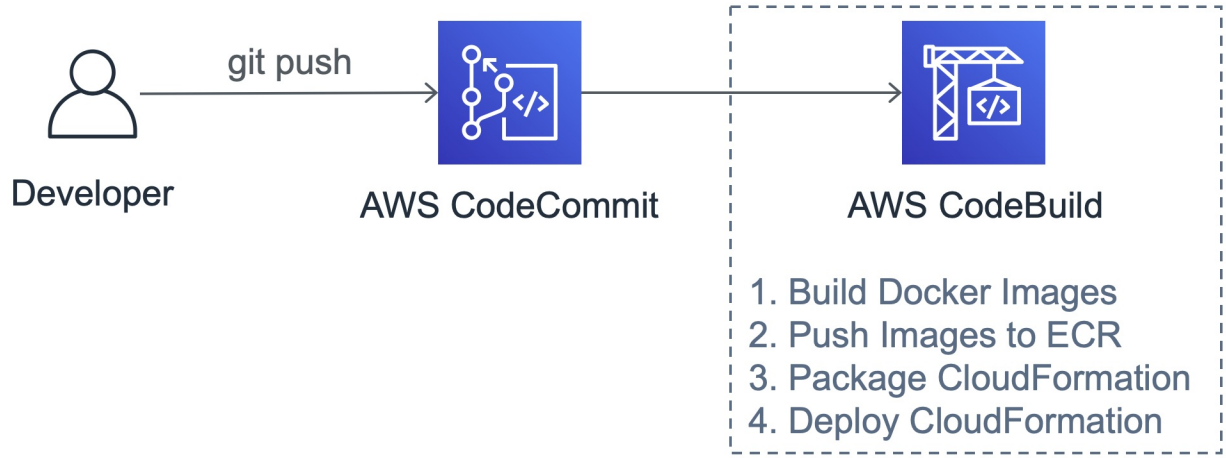


Figure 25: A change in the repository triggers a deployment pipeline

Benefits of a deployment pipeline

Often, a deployment process is manual, and only a few people on a team know the steps to deploy a new version of an application. Using a deployment pipeline empowers the whole team to push changes to production because everyone uses the same mechanism for deployments.

A deployment pipeline is an automated process, reducing the chance of human error, such as: missing a step, running steps out of order, and so on. If deployments are a big deal, they also cause stress on the person responsible for the deployment. With a deployment pipeline you will notice that deployments become routine for your team, which helps to reduce the stress level.

Last but not least, a deployment pipeline captures the experience of what the team has learned from numerous deployments in code instead of in a few brains. Every time a deployment does not work as expected, the pipeline is updated to handle the new edge case that no one foresaw.

Introducing CodeBuild

There are many options available when you are looking for ways to implement a deployment pipeline. You might have heard about Jenkins, CircleCi, BitBucket Pipelines, GitLab Pipelines, and many others. To keep things simple, we will use AWS CodeBuild, which comes with a superb AWS integration and which can be configured using CloudFormation.

AWS CodeBuild fetches code from CodeCommit and executes commands that you define in a file called `buildspec.yml` placed in the root directory of your project folder and repository. The following `buildspec.yml` changes the directory and runs the `npm i` command.

```
version: '0.2'
phases:
  build:
    commands:
    - 'cd aws'
    - 'npm i'
```

The `version` attribute specifies the schema of the `buildspec.yml` and is defined by AWS. Multiple `phases` are supported and run in the following order:

1. `install`
2. `pre_build` (if `install` succeeded)
3. `build` (if `pre_build` succeeded)
4. `post_build` (if `build` failed or succeeded)

Each phase runs a series of `commands` — defined by you — using the same instance of the default shell (we will use Bash) as the execution environment. Besides the environment variables that CodeBuild provides out of the box, you can also define custom environment variables to keep your `buildspec.yml` more flexible.

Deployment steps defined in buildspec.yml

The following steps are needed to deploy our sample application from chapter 2:

1. Build the nginx Docker image
2. Build the php-fpm Docker image

3. Push both Docker images to ECR
4. Install cfn-modules by executing `npm i` in the `aws` folder
5. Package the CloudFormation template (`aws cloudformation package`)
6. Deploy the CloudFormation template with the new Docker images (`aws cloudformation deploy`)

The following `buildspec.yml` installs Docker, executes the `docker login` command returned by `aws ecr get-login` and finally, runs the Bash script: `build.sh`. The Bash script is the place for your own customizations and helps us to keep the `buildspec.yml` clear.

```
version: '0.2'
phases:
  install:
    'runtime-versions':
      docker: 18
  pre_build:
    commands:
    - '$(aws ecr get-login --no-include-email)''
  build:
    commands:
    - 'bash build.sh'
```

Let's look at `build.sh` to see how steps 1-6 are implemented. The `build.sh` script is invoked by the `buildspec.yml` with `bash build.sh`. Therefore, the script lives side-by-side with the `buildspec.yml` file. The first two lines state that this is a Bash script that ends execution if one of the commands fails.

```
#!/bin/bash
set -e
```

The next lines create the names and tags for the Docker images:

```
NGINX_NAME_TAG="${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/\
${NGINX_REPO_NAME}:${CODEBUILD_RESOLVED_SOURCE_VERSION}"
FPM_NAME_TAG="${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/\
${FPM_REPO_NAME}:${CODEBUILD_RESOLVED_SOURCE_VERSION}"
```

Next, the script executes `docker build` to build the two Docker images, followed by `docker push` to upload the Docker images to ECR. Depending on the web application that you want to dockerize, you might want to add additional commands here (e.g., producing a JAR file, downloading dependencies, running unit tests or static analysis).

```
docker build -t "${NGINX_NAME_TAG}" \
  -f docker/nginx/Dockerfile .
docker build -t "${FPM_NAME_TAG}" \
  -f docker/php-fpm/Dockerfile .
docker push "${NGINX_NAME_TAG}"
docker push "${FPM_NAME_TAG}"
```

After that, we need to change to the aws directory, which contains the Infrastructure as Code, install the cfn-modules, and package the CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file template.yml \
  --s3-bucket "${BUCKET_NAME}" --output-template-file .template.yml
```

Finally, the CloudFormation template is deployed and the new Docker images are passed as parameters:

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name php-basic --capabilities CAPABILITY_IAM \
  --parameter-overrides "AppImage=${FPM_NAME_TAG}" \
"ProxyImage=${NGINX_NAME_TAG}"
```

You can find the complete `buildspec.yml` and `build.sh` files in the book's source code directory inside `php-basic-pipeline`.

Creating a CodeBuild project

So far, we have transformed the manual deployment steps into a repeatable Shell script. Next, you'll configure CodeBuild to connect the dots. A CodeBuild project fetches code from CodeCommit and executes the commands defined in the buildspec.yml. Whenever you push a new commit to your Git repository, CodeBuild will start automatically. The following snippet shows how to create a CodeBuild project with CloudFormation. Doing so allows you to use Infrastructure as Code to set up your deployment pipeline as well. More detailed explanation follows.

```
Resources:
  [...]
  Project:
    Type: 'AWS::CodeBuild::Project'
    Properties:
      Artifacts:
```

```
    Type: NO_ARTIFACTS
  Environment:
    ComputeType: BUILD_GENERAL1_SMALL
    EnvironmentVariables:
    - Name: ACCOUNT_ID
      Type: PLAINTEXT
      Value: !Ref 'AWS::AccountId'
    - Name: REGION
      Type: PLAINTEXT
      Value: !Ref 'AWS::Region'
    - Name: NGINX_REPO_NAME
      Type: PLAINTEXT
      Value: !Ref RepositoryNginx
    - Name: FPM_REPO_NAME
      Type: PLAINTEXT
      Value: !Ref RepositoryFpm
    - Name: BUCKET_NAME
      Type: PLAINTEXT
      Value: !Ref BucketArtifacts
    Image: 'aws/codebuild/standard:2.0'
    PrivilegedMode: true # required to build Docker images
    Type: LINUX_CONTAINER
  LogsConfig:
    CloudWatchLogs:
      GroupName: !Ref ProjectLogGroup
      Status: ENABLED
  ServiceRole: !GetAtt 'ProjectRole.Arn'
  Source:
    Location: !Sub >
      https://git-codecommit.${AWS::Region}.amazonaws.com/
      v1/repos/${CodeCommitRepositoryName}
    Type: CODECOMMIT
  TimeoutInMinutes: 30
```

The important pieces of the CloudFormation snippet are:

- The `Environment` property, which defines the custom environment variables that are used to keep the `buildspec.yml` file more flexible. Add more variables here if you need them in your modified `build.sh` script.
- The `LogsConfig` property, which specifies the CloudWatch Logs location where logs are stored that you can use to debug problems with the pipeline.
- The `ServiceRole` property, which points to the IAM role that is used when executing `buildspec.yml`. The role needs permissions to download the source code, write to CloudWatch Logs, deploy the CloudFormation template, and push images to ECR. If you want to call other AWS services in your `build.sh` script, you likely need to add additional permissions here.

You can find the full `pipeline.yml` template in the book's source code directory inside `php-basic-pipeline/aws`.

Deploying the deployment pipeline

> **Dependency** The prepared deployment pipeline works together with the `php-basic` application from chapter 2 and requires that you finish section 4.1.

Execute the following commands in your temporary working environment to deploy the deployment pipeline. First, copy the files into the example application folder from chapter 2:

```
cd /src/php-basic
cp -R ../php-basic-pipeline/* .
```

Commit the new files to your local Git repository:

```
git add -A
git commit -m 'add deployment pipeline'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name php-basic-pipeline \
  --capabilities CAPABILITY_IAM
```

The last command pushes a change to the remote repository which then triggers CodeBuild to run:

```
git push
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `php-basic-pipeline`.

3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Use the following command to retrieve the URL of your web application running on AWS and open it in your browser:

```
aws cloudformation describe-stacks --stack-name php-basic \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

Now, make a change to the index.php file and push that change:

```
cd /src/php-basic
nano index.php
git add index.php
git commit -m 'deployment test'
git push
```

You will soon see a new CodeBuild run in the Management Console. Reload the web application after the build completes.

That's it. Your deployment pipeline will revolutionize the way you develop and ship software to your users!

You have reached the end of the chapter. Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/php-basic/aws
bash pipeline-cleanup.sh
```

Is your deployment pipeline not working? Please share your problem with us and the community: https://community.cloudonaut.io.

Appendix: Example web applications

Dependency To understand the examples in the appendix, please work through chapters one, two, and four first.

The following example web applications are available:

- Java Spring Boot: Java web application based on the Spring Boot framework and Maven
- Node.js Express: Node.js web application based on the Express framework
- PHP Apache: PHP web application hosted by Apache
- Python Django: Python application based on the Django framework
- Ruby on Rails: Ruby application based on the Rails framework

---

A short reminder of how to start your temporary working environment:

On Mac and Linux

Open a Terminal and change into the book's source code directory:

```
cd docker-on-aws-code
```

Run the following command to start your temporary working environment:

```
bash start.sh
```

On Windows

Open PowerShell and change into the book's source code directory:

```
cd docker-on-aws-code
```

Run the following command to start your temporary working environment:

```
powershell -ExecutionPolicy ByPass -File start.ps1
```

Java Spring Boot

Spring Boot provides a rapid way to create and run Java applications on top of the popular Spring Framework. We prepared an example web application to connect the dots for you. The example application is a port of the php-basic application from chapter 2. Let's dive into the example application.

Building the Docker images

In your temporary working environment, change directory:

```
cd /src/java-spring-boot
```

Java is not a scripting language like PHP. You cannot execute Java code directly. Instead, the Java source code needs to be compiled into bytecode, which can be executed by the Java Virtual Machine. A popular tool to manage this process is Maven. Maven manages dependencies, turns Java source code into bytecode, packages Java Archives (JARs), and much more.

The following command downloads all dependencies, compiles the source code, and generates a JAR file which contains the runnable application:

```
mvn package
```

Once you have generated the JAR file, you can try to start the application:

```
java -jar target/rapid-docker-on-aws-1.0.0.jar
```

You will see some errors because the application cannot connect to a database, which is fine at the moment. Before we fix the missing database, we inspect the `Dockerfile`. A two container approach is used here. NGINX serves the static files and redirects the dynamic pages to a second container that runs the Spring Boot server.

The NGINX `Dockerfile` is exactly the same as in chapter 2. The only difference is the NGINX configuration file, which forwards requests for HTML files to the Spring Boot server.

```
server {
```

```
    listen       80;
    server_name  localhost;
    root         /var/www/html;
    # pass the HTML requests to Spring Boot server listening on 127.
    location ~ \.html$ {
        proxy_pass   http://127.0.0.1:8080;
    }
    # redirect / to index.html
    location ~ ^\/$ {
        return 301 $scheme://$http_host/index.html$is_args$args;
    }
}
```

The Spring Boot server is already embedded into the JAR file. We only need a Java Runtime Environment (JRE) to execute the JAR file. The following `Dockerfile`:

1. Is based on Amazon Linux 2.
2. Installs Amazon Corretto - a production-ready distribution of OpenJDK, which provides a JRE.
3. Installs wait-for-it - a helper to make sure that the MySQL database is up and running before the Java container is started with `docker-compose`.
4. Copies the JAR file.
5. Exposes port 8080 and defines the default command to run the JAR.

> **Info** `envsubst` is not needed because Spring Boot comes with it's own mechanism for config files that supports environment variables out of the box.

```
FROM amazonlinux:2.0.20190508

# Install JDK 1.8
RUN amazon-linux-extras install corretto8

# Install wait-for-it
COPY docker/wait-for-it.sh /usr/local/bin/
RUN chmod u+x /usr/local/bin/wait-for-it.sh

# Copy JAR
COPY target/rapid-docker-on-aws-1.0.0.jar /opt/

# Expose port 8080 and start Spring Boot server
EXPOSE 8080
CMD ["java", "-XX:+ExitOnOutOfMemoryError", \
  "-Djava.net.preferIPv4Stack=true", "-jar", \
```

```
  "/opt/rapid-docker-on-aws-1.0.0.jar"]
```

Testing locally

You can run the application locally using Docker Compose:

```
docker-compose -f docker/docker-compose.yml up --build
```

Magically, Docker Compose will spin up three containers. Point your browser to http://localhost:8080 to check that your web application is up and running. The log files of all containers will show up in your terminal, which simplifies debugging a lot.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers:

```
docker-compose -f docker/docker-compose.yml down
```

Deploying on AWS

You can also deploy the web application on AWS. If you have not changed any code, you can skip steps 1-5.

(1) Build Docker images:

```
docker build -t java-spring-boot-nginx:latest \
  -f docker/nginx/Dockerfile .
docker build -t java-spring-boot-java:latest \
  -f docker/java/Dockerfile .
```

(2) Create ECR repositories:

```
aws ecr create-repository \
  --repository-name java-spring-boot-nginx \
  --query 'repository.repositoryUri' --output text
aws ecr create-repository --repository-name java-spring-boot-java \
  --query 'repository.repositoryUri' --output text
```

(3) Login to Docker registry (ECR):

```
$(aws ecr get-login --no-include-email)
```

(4) Tag Docker images:

```
docker tag java-spring-boot-nginx:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-nginx:latest
docker tag java-spring-boot-java:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-java:latest
```

(5) Push Docker images:

```
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-nginx:latest
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-java:latest
```

(6) Install cfn-modules:

```
cd aws
npm i
```

(7) Package CloudFormation template:

```
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

(8) Deploy CloudFormation stack (leave out the `--parameter-overrides` if you skipped steps 1-5):

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name java-spring-boot --capabilities CAPABILITY_IAM \
  --parameter-overrides \
AppImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-java:latest \
ProxyImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
java-spring-boot-nginx:latest
```

(9) Retrieve the application's URL:

```
aws cloudformation describe-stacks --stack-name java-spring-boot \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

> **Note** When started, the Java container takes more than 60 seconds to answer the first requests. Therefore, the parameter `HealthCheckGracePeriodSeconds` is set to 120 seconds to ignore health checks in the first two minutes instead of only the first minute.

Don't forget to clean up after you're finished testing:

```
cd /src/java-spring-boot/aws/
bash cleanup.sh
```

Deploying continuously

First, you have to create a CodeCommit repository for the application. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name java-spring-boot
```

The output will be similar to the JSON below. You will need the value of the cloneUrlHttp attribute later.

```
{
  "repositoryMetadata": {
    "repositoryName": "java-spring-boot",
    "cloneUrlHttp": "https://git-codecommit[...]/java-spring-boot",
    [...]
  }
}
```

Inside the `java-spring-boot` directory, create a Git repository and add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/java-spring-boot
git init
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/java-spring-boot
```

Add the pipeline files into the application's folder next:

```
cp -R ../java-spring-boot-pipeline/* .
```

Afterwards, commit all files and push them to the remote repository:

```
git add -A
git commit -m 'initial commit'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name java-spring-boot-pipeline \
  --capabilities CAPABILITY_IAM
```

Pushing your local changes to the remote repository will start the deployment pipeline:

```
git push -u origin master
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `java-spring-boot-pipeline`.
3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/java-spring-boot/aws
bash pipeline-cleanup.sh
```

Node.js Express

Express provides a rapid way to create Node.js applications. We prepared an example web application to connect the dots for you. The example application is a port of the php-basic application from chapter 2. Let's dive into the example application.

Building the Docker image

In your temporary working environment, change directory:

```
cd /src/nodejs-express
```

The `Dockerfile` is based on the official Node.js Docker Image: `node:10.16.2-stretch`. Static files (folders `img` and `css`) are served by Express instead of a separate NGINX container. The following details are required to understand the `Dockerfile`:

- `envsubst` is used to generate the config file from environment variables
- `npm ci --only=production` installs the dependencies declared in `package.json` (`package-lock.json`, to be more precise)
- The Express application listens on port 8080
- The Express application's entry point is `server.js` and can be started with `node server.js`

```
FROM node:10.16.2-stretch

WORKDIR /usr/src/app

ENV NODE_ENV production

# Install envsubst
RUN apt-get update && apt-get install -y gettext
COPY docker/custom-entrypoint /usr/local/bin/
RUN chmod u+x /usr/local/bin/custom-entrypoint
ENTRYPOINT ["custom-entrypoint"]
RUN mkdir /usr/src/app/config/

# Copy config files
COPY config/*.tmp /tmp/config/
```

```
# Install Node.js dependencies
COPY package*.json /usr/src/app/
RUN npm ci --only=production

# Copy Node.js files
COPY css /usr/src/app/css
COPY img /usr/src/app/img
COPY views /usr/src/app/views
COPY server.js /usr/src/app/

# Expose port 8080 and start Node.js server
EXPOSE 8080
CMD ["node", "server.js"]
```

Testing locally

You can run the application locally using Docker Compose:

```
docker-compose -f docker/docker-compose.yml up --build
```

Magically, Docker Compose will spin up two containers. Point your browser to http://localhost:8080 to check that your web application is up and running. The log files of all containers will show up in your terminal, which simplifies debugging a lot.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers:

```
docker-compose -f docker/docker-compose.yml down
```

Deploying on AWS

You can also deploy the web application on AWS. If you have not changed any code, you can skip steps 1-5.

(1) Build Docker image:

```
docker build -t nodejs-express:latest -f docker/Dockerfile .
```

(2) Create ECR repository:

```
aws ecr create-repository --repository-name nodejs-express \
```

```
  --query 'repository.repositoryUri' --output text
```

(3) Login to Docker registry (ECR):

```
$(aws ecr get-login --no-include-email)
```

(4) Tag Docker image:

```
docker tag nodejs-express:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
nodejs-express:latest
```

(5) Push Docker image:

```
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
nodejs-express:latest
```

(6) Install cfn-modules:

```
cd aws
npm i
```

(7) Package CloudFormation template:

```
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

(8) Deploy CloudFormation stack (leave out the `--parameter-overrides` if you skipped steps 1-5):

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name nodejs-express --capabilities CAPABILITY_IAM \
  --parameter-overrides \
AppImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
nodejs-express:latest
```

(9) Retrieve the application's URL:

```
aws cloudformation describe-stacks --stack-name nodejs-express \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

Don't forget to clean up after you're finished testing:

```
cd /src/nodejs-expresse/aws/
bash cleanup.sh
```

Deploying continuously

First, you have to create a CodeCommit repository for the application. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name nodejs-express
```

The output will be similar to the JSON below. You will need the value of the cloneUrlHttp attribute later.

```
{
  "repositoryMetadata": {
    "repositoryName": "nodejs-express",
    "cloneUrlHttp": "https://git-codecommit[...]/nodejs-express",
    [...]
  }
}
```

Inside the `nodejs-express` directory, create a Git repository and add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/nodejs-express
git init
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/nodejs-express
```

Add the pipeline files into the application's folder next:

```
cp -R ../nodejs-express-pipeline/* .
```

Afterwards, commit all files and push them to the remote repository:

```
git add -A
```

```
git commit -m 'initial commit'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name nodejs-express-pipeline \
  --capabilities CAPABILITY_IAM
```

Pushing your local changes to the remote repository will start the deployment pipeline:

```
git push -u origin master
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `nodejs-express-pipeline`.
3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/nodejs-express/aws
bash pipeline-cleanup.sh
```

PHP Apache

The Apache HTTP Server is a popular alternative to NGINX. Remember that the PHP-FPM + NGINX example (php-basic) uses two containers (one for NGINX one for PHP-FPM). The way the Apache HTTP server serves PHP pages is slightly different. PHP is embedded into the Apache HTTP server as a module. Therefore, the PHP interpreter runs inside the Apache process. You might ask yourself: Should I use NGINX or Apache? We believe that the NGINX based setup is very efficient while the Apache based setup is more flexibel and a good fit for beginners. Let's dive into the example application.

Building the Docker image

In your temporary working environment, change directory:

```
cd /src/php-apache
```

The `Dockerfile` is no longer based on `php:7.3-fpm-stretch` from chapter 2. Instead, we use `php:7.3-apache-stretch`, which includes the Apache HTTP Server. Static files (folders `img` and `css`) are now served by Apache instead of a separate NGINX container. The apache process runs as user `www-data`. Therefore, the files are now owned by that user.

```
FROM php:7.3-apache-stretch

# Configure PHP
RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"
RUN docker-php-ext-install -j$(nproc) pdo pdo_mysql

# Install envsubst
RUN apt-get update && apt-get install -y gettext
COPY docker/custom-entrypoint /usr/local/bin/
RUN chmod u+x /usr/local/bin/custom-entrypoint
ENTRYPOINT ["custom-entrypoint"]
RUN mkdir /var/www/html/conf/

# Copy config files
COPY conf/*.tmp /tmp/conf/

# Copy PHP and static files
COPY css /var/www/html/css
COPY img /var/www/html/img
```

```
COPY *.php /var/www/html/
COPY lib /var/www/html/lib
RUN chown -R www-data:www-data /var/www/html

# Start Apache server
CMD ["apache2-foreground"]
```

Testing locally

You can run the application locally using Docker Compose:

```
docker-compose -f docker/docker-compose.yml up --build
```

Magically, Docker Compose will spin up two containers. Point your browser to http://localhost:8080 to check that your web application is up and running. The log files of all containers will show up in your terminal, which simplifies debugging a lot.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers:

```
docker-compose -f docker/docker-compose.yml down
```

Deploying on AWS

You can also deploy the web application on AWS. If you have not changed any code, you can skip steps 1-5.

(1) Build Docker image:

```
docker build -t php-apache:latest -f docker/Dockerfile .
```

(2) Create ECR repository:

```
aws ecr create-repository --repository-name php-apache \
  --query 'repository.repositoryUri' --output text
```

(3) Login to Docker registry (ECR):

```
$(aws ecr get-login --no-include-email)
```

(4) Tag Docker image:

```
docker tag php-apache:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-apache:latest
```

(5) Push Docker image:

```
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-apache:latest
```

(6) Install cfn-modules:

```
cd aws
npm i
```

(7) Package CloudFormation template:

```
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

(8) Deploy CloudFormation stack (leave out the `--parameter-overrides` if you skipped steps 1-5):

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name php-apache --capabilities CAPABILITY_IAM \
  --parameter-overrides \
AppImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
php-apache:latest
```

(9) Retrieve the application's URL:

```
aws cloudformation describe-stacks --stack-name php-apache \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

Don't forget to clean up after you're finished testing:

```
cd /src/php-apache/aws/
bash cleanup.sh
```

Deploying continuously

First, you have to create a CodeCommit repository for the application. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name php-apache
```

The output will be similar to the JSON below. You will need the value of the cloneUrlHttp attribute later.

```
{
  "repositoryMetadata": {
    "repositoryName": "php-apache",
    "cloneUrlHttp": "https://git-codecommit[...]/php-apache",
    [...]
  }
}
```

Inside the `php-apache` directory, create a Git repository and add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/php-apache
git init
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/php-apache
```

Add the pipeline files into the application's folder next:

```
cp -R ../php-apache-pipeline/* .
```

Afterwards, commit all files and push them to the remote repository:

```
git add -A
git commit -m 'initial commit'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name php-apache-pipeline --capabilities CAPABILITY_IAM
```

Pushing your local changes to the remote repository will start the deployment pipeline:

```
git push -u origin master
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `php-apache-pipeline`.
3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/php-apache/aws
bash pipeline-cleanup.sh
```

Python Django

Django is a popular Python web framework that encourages rapid development and clean, pragmatic design. We prepared an example web application to connect the dots for you. The example application is a port of the php-basic application from chapter 2. Let's dive into the example application.

Building the Docker images

In your temporary working environment, change directory:

```
cd /src/python-django
```

The example is using two Docker images:

- NGINX to serve static files and proxy to Django
- Python Django application

The NGINX `Dockerfile` makes use of multi-stage builds. You can use more than one `FROM` statements in your Dockerfile. In this case:

1. Static assets are generated in a Python stage
    1. Based on the official python image
    2. Using `pip3` to install the Python dependencies
    3. Copying the app
    4. Generating the static assets with `python3 manage.py collectstatic` (output goes to the `assets` folder)
2. The static assets are copied (using `COPY --from=build`) into the NGINX stage which produces the final Docker image
    1. Based on the official nginx image
    2. Copying the `static/` folder from the previous stage

```
# Static Assets
FROM python:3.7.4 AS build

WORKDIR /usr/src/app

# Install Python dependencies
COPY requirements.txt /usr/src/app/
RUN pip3 install -r requirements.txt
```

```
# Copy Python files
COPY example /usr/src/app/example
COPY rapid /usr/src/app/rapid
COPY manage.py /usr/src/app/

# Build static assets
RUN SECRET_KEY=secret python3 manage.py collectstatic



# NGINX
FROM nginx:1.14

# Configure NGINX
COPY docker/nginx/default.conf /etc/nginx/conf.d/default.conf

# Copy static files
COPY --from=build /usr/src/app/build/ /var/www/html/static
RUN chown -R nginx:nginx /var/www/html
```

The NGINX configuration file forwards requests to the Python container if the path does not start with `/static/`.

```
server {
    listen       80;
    server_name  localhost;
    root         /var/www/html;

    location ~ ^/static/ {
      # serve from NGINX
    }

    location / {
      # pass to Python gunicorn based on
      # http://docs.gunicorn.org/en/stable/deploy.html
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
      proxy_set_header Host $http_host;
      # we don't want nginx trying to do something clever with
      # redirects, we set the Host: header above already.
      proxy_redirect off;
      proxy_pass http://127.0.0.1:8000;
    }
}
```

The Python Django `Dockerfile` is based on the official python image, uses `pip3` to install the Python dependencies and uses gunicorn to run the app.

Info `envsubst` is not needed because Django comes with it's own mechanism for config files that supports environment variables out of the box.

```
FROM python:3.7.4

WORKDIR /usr/src/app

# Install wait-for-it
COPY docker/wait-for-it.sh /usr/local/bin/
RUN chmod u+x /usr/local/bin/wait-for-it.sh

# Install Python dependencies
COPY requirements.txt /usr/src/app/
RUN pip3 install -r requirements.txt

# Copy Python files
COPY example /usr/src/app/example
COPY rapid /usr/src/app/rapid
COPY manage.py /usr/src/app/

# Configure custom etrypoint to run migrations
COPY docker/python/custom-entrypoint /usr/local/bin/
RUN chmod u+x /usr/local/bin/custom-entrypoint
ENTRYPOINT ["custom-entrypoint"]

# Expose port 8000 and start Python server
EXPOSE 8000
CMD ["gunicorn", "-b", "0.0.0.0", "-w", "2", "rapid.wsgi"]
```

Customization The `-w` parameter of gunicorn defined the number of workers and should be in the range of `2-4 x $(NUM_CORES)`. If you change the `Cpu` property in the CloudFormation template you have to modify `-w` as well.

Testing locally

You can run the application locally using Docker Compose:

```
docker-compose -f docker/docker-compose.yml up --build
```

Magically, Docker Compose will spin up three containers. Point your browser to http://localhost:8080 to check that your web application is up and running. The log files of all containers will show up in your terminal, which simplifies debugging a lot.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers:

```
docker-compose -f docker/docker-compose.yml down
```

Deploying on AWS

You can also deploy the web application on AWS. If you have not changed any code, you can skip steps 1-5.

(1) Build Docker images:

```
docker build -t python-django-nginx:latest \
  -f docker/nginx/Dockerfile .
docker build -t python-django-python:latest \
  -f docker/python/Dockerfile .
```

(2) Create ECR repositories:

```
aws ecr create-repository --repository-name python-django-nginx \
  --query 'repository.repositoryUri' --output text
aws ecr create-repository --repository-name python-django-python \
  --query 'repository.repositoryUri' --output text
```

(3) Login to Docker registry (ECR):

```
$(aws ecr get-login --no-include-email)
```

(4) Tag Docker images:

```
docker tag python-django-nginx:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
python-django-nginx:latest
docker tag python-django-python:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
python-django-python:latest
```

(5) Push Docker images:

```
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
python-django-nginx:latest
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
```

```
python-django-python:latest
```

(6) Install cfn-modules:

```
cd aws
npm i
```

(7) Package CloudFormation template:

```
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

(8) Deploy CloudFormation stack (leave out the `--parameter-overrides` if you skipped steps 1-5):

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name python-django --capabilities CAPABILITY_IAM \
  --parameter-overrides \
AppImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
python-django-python:latest \
ProxyImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
python-django-nginx:latest
```

(9) Retrieve the application's URL:

```
aws cloudformation describe-stacks --stack-name python-django \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

Don't forget to clean up after you're finished testing:

```
cd /src/python-django/aws/
bash cleanup.sh
```

Deploying continuously

First, you have to create a CodeCommit repository for the application. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name python-django
```

The output will be similar to the JSON below. You will need the value of the

cloneUrlHttp attribute later.

```
{
  "repositoryMetadata": {
    "repositoryName": "python-django",
    "cloneUrlHttp": "https://git-codecommit[...]/python-django",
    [...]
  }
}
```

Inside the `python-django` directory, create a Git repository and add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/python-django
git init
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/python-django
```

Add the pipeline files into the application's folder next:

```
cp -R ../python-django-pipeline/* .
```

Afterwards, commit all files and push them to the remote repository:

```
git add -A
git commit -m 'initial commit'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name python-django-pipeline \
  --capabilities CAPABILITY_IAM
```

Pushing your local changes to the remote repository will start the deployment pipeline:

```
git push -u origin master
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `python-django-pipeline`.
3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/python-django/aws
bash pipeline-cleanup.sh
```

Ruby on Rails

Ruby on Rails is a popular web-application framework for Ruby. We prepared an example web application to connect the dots for you. The example application is a port of the php-basic application from chapter 2. Let's dive into the example application.

Building the Docker image

In your temporary working environment, change directory:

```
cd /src/ruby-rails
```

Ruby on Rails runs by default on Puma - A Ruby/Rack web server built for concurrency. The following `Dockerfile`:

1. Is based on Amazon Linux 2.
2. Installs Node.js, required by yarn, required by Ruby on Rails.
3. Installs Ruby and Ruby on Rails with all needed dependencies.
4. Installs wait-for-it - a helper to make sure that the MySQL database is up and running before the Ruby container is started with `docker-compose`.
5. Copies all files except the ignores defined in the file `.dockerignore`.
6. Installs all Ruby dependencies of the application and generates the static assets.
7. Configured a custom entry point that runs the database migrations when the container starts before the application is started.
8. Exposes port 3000 and defines the default command to run the application.

> **Info** `envsubst` is not needed because Ruby on Rails comes with it's own configuration mechanism that supports environment variables out of the box. Check out the file `config/database.rb` to see how environment variables are used to configure the database connection.

```
FROM amazonlinux:2.0.20190508

RUN mkdir /usr/src/app
WORKDIR /usr/src/app

# Install Node.js (needed for yarn)
```

```
RUN curl -sL https://rpm.nodesource.com/setup_10.x | bash -
RUN yum -y install nodejs gcc-c++ make

# Install Ruby & Rails
RUN curl -sL -o \
  /etc/yum.repos.d/yarn.repo https://dl.yarnpkg.com/rpm/yarn.repo
RUN amazon-linux-extras enable ruby2.6 \
  && yum -y install git tar gzip yarn zlib-devel sqlite-devel \
  mariadb-devel ruby-devel rubygems-devel rubygem-bundler \
  rubygem-io-console rubygem-irb rubygem-json rubygem-minitest \
  rubygem-net-http-persistent rubygem-net-telnet \
  rubygem-power_assert rubygem-rake rubygem-test-unit \
  rubygem-thor rubygem-xmlrpc \
  && gem install rails

# Install wait-for-it
COPY docker/wait-for-it.sh /usr/local/bin/
RUN chmod u+x /usr/local/bin/wait-for-it.sh

# Copy Ruby files (see .dockerignore)
COPY . .

# Install Ruby dependencies
ENV RAILS_ENV production
ENV RAILS_LOG_TO_STDOUT enabled
ENV RAILS_SERVE_STATIC_FILES enabled
RUN bin/bundle install --deployment --without development test
# see https://github.com/rails/rails/issues/32947 for workaround
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

# Configure custom etrypoint to run migrations
COPY docker/custom-entrypoint /usr/local/bin/
RUN chmod u+x /usr/local/bin/custom-entrypoint
ENTRYPOINT ["custom-entrypoint"]

# Expose port 3000 and start Rails server
EXPOSE 3000
CMD ["bin/rails", "server", "--binding=0.0.0.0"]
```

Testing locally

You can run the application locally using Docker Compose:

```
docker-compose -f docker/docker-compose.yml up --build
```

Magically, Docker Compose will spin up two containers. Point your browser to http://localhost:8080 to check that your web application is up and running. The

log files of all containers will show up in your terminal, which simplifies debugging a lot.

After you have verified that your application is working correctly, cancel the running `docker-compose` process by pressing `CTRL + C`, and tear down the containers:

```
docker-compose -f docker/docker-compose.yml down
```

Deploying on AWS

You can also deploy the web application on AWS. If you have not changed any code, you can skip steps 1-5.

(1) Build Docker image:

```
docker build -t ruby-rails:latest -f docker/Dockerfile .
```

(2) Create ECR repository:

```
aws ecr create-repository --repository-name ruby-rails \
  --query 'repository.repositoryUri' --output text
```

(3) Login to Docker registry (ECR):

```
$(aws ecr get-login --no-include-email)
```

(4) Tag Docker image:

```
docker tag ruby-rails:latest \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
ruby-rails:latest
```

(5) Push Docker image:

```
docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
ruby-rails:latest
```

(6) Install cfn-modules:

```
cd aws
npm i
```

(7) Package CloudFormation template:

```
aws cloudformation package --template-file template.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .template.yml
```

(8) Deploy CloudFormation stack (leave out the `--parameter-overrides` if you skipped steps 1-5):

```
aws cloudformation deploy --template-file .template.yml \
  --stack-name ruby-rails --capabilities CAPABILITY_IAM \
  --parameter-overrides \
AppImage=111111111111.dkr.ecr.eu-west-1.amazonaws.com/\
ruby-rails:latest
```

(9) Retrieve the application's URL:

```
aws cloudformation describe-stacks --stack-name ruby-rails \
  --query 'Stacks[0].Outputs[?OutputKey==`Url`].OutputValue' \
  --output text
```

> **Note** When started, the Ruby on Rails container takes more than 60 seconds to answer the first requests. Therefore, the parameter `HealthCheckGracePeriodSeconds` is set to 120 seconds to ignore health checks in the first two minutes instead of only the first minute.

Don't forget to clean up after you're finished testing:

```
cd /src/ruby-rails/aws/
bash cleanup.sh
```

Deploying continuously

First, you have to create a CodeCommit repository for the application. In the temporary working environment, execute the following command to create the repository:

```
aws codecommit create-repository --repository-name ruby-rails
```

The output will be similar to the JSON below. You will need the value of the cloneUrlHttp attribute later.

```
{
```

```
  "repositoryMetadata": {
    "repositoryName": "ruby-rails",
    "cloneUrlHttp": "https://git-codecommit[...]/ruby-rails",
    [...]
  }
}
```

Inside the `ruby-rails` directory, create a Git repository and add the newly created remote repository. Replace `$REPO_URL` with the `cloneUrlHttp` output from the above command:

```
cd /src/ruby-rails
git init
git config credential.helper '!aws codecommit credential-helper $@'
git config credential.UseHttpPath true
git remote add origin $REPO_URL
```

For example, the last command may look like this:

```
git remote add origin https://git-codecommit.\
eu-west-1.amazonaws.com/v1/repos/ruby-rails
```

Add the pipeline files into the application's folder next:

```
cp -R ../ruby-rails-pipeline/* .
```

Afterwards, commit all files and push them to the remote repository:

```
git add -A
git commit -m 'initial commit'
```

Now, it's time to deploy the CodeBuild project using the prepared CloudFormation template:

```
cd aws
npm i
aws cloudformation package --template-file pipeline.yml \
  --s3-bucket docker-on-aws-$NICKNAME \
  --output-template-file .pipeline.yml
aws cloudformation deploy --template-file .pipeline.yml \
  --stack-name ruby-rails-pipeline --capabilities CAPABILITY_IAM
```

Pushing your local changes to the remote repository will start the deployment pipeline:

```
git push -u origin master
```

To observe what happens behind the scenes:

1. Visit the CodeBuild Management Console (you might need to set the AWS region at the very top right)
2. Click on the CodeBuild project named `ruby-rails-pipeline`.
3. Under **Build history**, you will see a build run.
4. Click on the build run.
5. You will see the logs of the build.
6. Wait until the build run has finished.

The first run might take up to 20 minutes. Further runs will be much faster.

Don't forget to delete the AWS infrastructure and repositories to avoid unnecessary costs:

```
cd /src/ruby-rails/aws
bash pipeline-cleanup.sh
```

Table of Contents