

# Securing DevOps

Security in the cloud

Julien Vehent

**Sample Chapter**

 MANNING



# ***Securing DevOps***

by Julien Vehent

## **Chapter 2**

Copyright 2018 Manning Publications

# *brief contents*

---

	1 ■ Securing DevOps	1
<b>PART 1</b>	<b>CASE STUDY: APPLYING LAYERS OF SECURITY TO A SIMPLE DEVOPS PIPELINE.....</b>	<b>19</b>
	2 ■ Building a barebones DevOps pipeline	21
	3 ■ Security layer 1: protecting web applications	45
	4 ■ Security layer 2: protecting cloud infrastructures	78
	5 ■ Security layer 3: securing communications	119
	6 ■ Security layer 4: securing the delivery pipeline	148
<b>PART 2</b>	<b>WATCHING FOR ANOMALIES AND PROTECTING SERVICES AGAINST ATTACKS .....</b>	<b>177</b>
	7 ■ Collecting and storing logs	179
	8 ■ Analyzing logs for fraud and attacks	208
	9 ■ Detecting intrusions	240
	10 ■ The Caribbean breach: a case study in incident response	275
<b>PART 3</b>	<b>MATURING DEVOPS SECURITY .....</b>	<b>299</b>
	11 ■ Assessing risks	301
	12 ■ Testing security	329
	13 ■ Continuous security	354

# *Building a barebones DevOps pipeline*

---

## ***This chapter covers***

- Configuring a CI pipeline for an example invoicer application
- Deploying the invoicer in AWS
- Identifying areas of a DevOps pipeline that require security attention

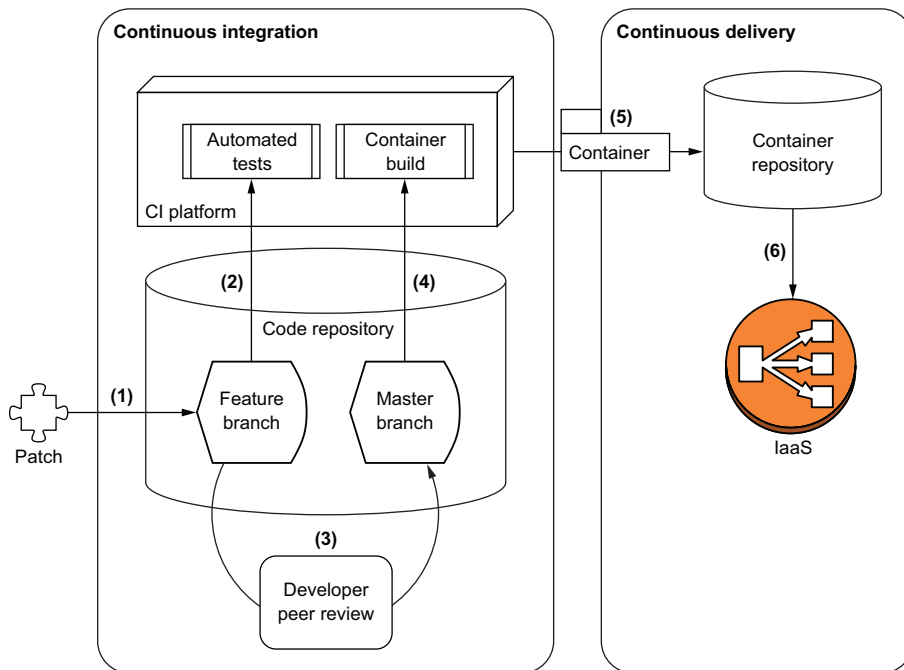
In chapter 1, I outlined an ambitious security strategy and described why security must be an integral component of the product. For security to be a part of DevOps, we must first understand how applications are built, deployed, and operated in DevOps. We'll ignore security in this chapter and focus on building a fully functional DevOps pipeline to understand the techniques of DevOps and set the stage for security discussions we'll have in chapters 3, 4, and 5.

DevOps is more about concepts, ideas, and workflows than it is about recommending one specific technology. A DevOps standard may not exist, yet it has consistent patterns across implementations. In this chapter, we take a specific example to implement those patterns: the invoicer, a small web API that manages invoices through a handful of HTTP endpoints. It's written in Go and its source code is available at <https://securing-devops.com/ch02/invoicer>.

## 2.1 Implementation roadmap

We want to manage and operate the invoicer the DevOps way. To achieve this, we'll implement the various steps of CI, CD, and IaaS that will allow us to quickly release and deploy new versions of the software to our users. Our goal is to go from patch submission to deploying in production in under 15 minutes with a mostly automated process. The pipeline you'll build is described in figure 2.1 and is composed of six steps:

- 1 A developer writes a patch and publishes it to a feature branch of the code repository.
- 2 Automated tests are run against the application.
- 3 A peer of the developer reviews the patch and merges it into the master branch of the code repository.
- 4 A new version of the application is automatically built and packaged into a container.
- 5 The container is published to a public registry.
- 6 The production infrastructure retrieves the container from the registry and deploys it.



**Figure 2.1** The complete CI/CD/IaaS pipeline to host the invoicer is composed of six steps that take a patch to a deployed application.

Building this pipeline requires integrating several components to work with each other. Your environment will need the following:

- *A source code repository*—Open source and proprietary solutions exist to manage source code: Bitbucket, Beanstalk, GitHub, GitLab, SourceForge, and so on. A popular choice at the time of writing is GitHub, which we'll use to host the invoicer's code.
- *A CI platform*—Again, the options are numerous: Travis CI, CircleCI, Jenkins, GitLab, and so on. Depending on your needs and environment, there's a CI platform for you. In this example, we'll use CircleCI because it integrates easily with GitHub and allows SSH access to build instances, which is handy for debugging the build steps.
- *A container repository*—The container world is evolving rapidly, but Docker is the standard choice at the time of writing. We'll use the repository provided by Docker Hub at [hub.docker.com](https://hub.docker.com).
- *An IaaS provider*—Google Cloud Platform and Amazon Web Services (AWS) are the two most popular IaaS providers at the time of writing. Some organizations prefer to self-host their IaaS and turn to solutions like Kubernetes or OpenStack to implement a layer of management on top of their own hardware (note that Kubernetes can also be used on top of EC2 instances in AWS). In this book, I use AWS because it's the most popular and mature IaaS on the market.

Let's summarize your toolkit: GitHub hosts the code and calls CircleCI when patches are sent. CircleCI builds the application into a container and pushes it to Docker Hub. AWS runs the infrastructure and retrieves new containers from Docker Hub to upgrade the production environment to the latest version. Simple, yet elegant.

### Every environment is different

It's unlikely that the environment your organization uses is an exact match with the one in this book, and some of the more specific security controls won't apply directly to the tools you use. This is expected, and I highlight security concepts before specific implementations, so you can transport them to your environment without too much trouble.

For example, the use of GitHub, Docker, or AWS may be disconcerting if your organization uses different tools. I use them as teaching tools, to explain the techniques of DevOps. Treat this chapter as a laboratory to learn and experiment with concepts, and then implement these concepts in whichever platform works best for you.

Keep in mind that even traditional infrastructures can benefit from modern DevOps techniques by building the exact same CI/CD/IaaS pipeline third-party tools provide, only internally. When you change technologies, the tools and terminology change, but the overall concepts, particularly the security ones, remain the same.

This pipeline uses tools and services that are available for free, at least long enough for you to follow along. The code and examples that follow are designed to be copied

and reused in order to build your own pipeline. Setting up your own environment is an excellent companion to reading this chapter.

## 2.2 *The code repository: GitHub*

When you head over to <https://securing-devops.com/ch02/invoicer>, you'll be redirected to the invoicer's GitHub repository. This repository hosts the source code of the invoicer application, as well as scripts that simplify the setup of the infrastructure. If you want to create your own version of the pipeline, *fork* the repository into your own account, which will copy Git files under your personal space, and follow the instructions in the README file to set up your environment. This chapter details all the steps to get your environment up and running, some of which are automated in scripts hosted in the repository.

## 2.3 *The CI platform: CircleCI*

In this section, you'll configure CircleCI to run tests and build a Docker container when changes are applied to the invoicer. The example in this section is specific to CircleCI, but the concept of using a CI platform to test and build an application is general and can easily be reproduced in other CI platforms.

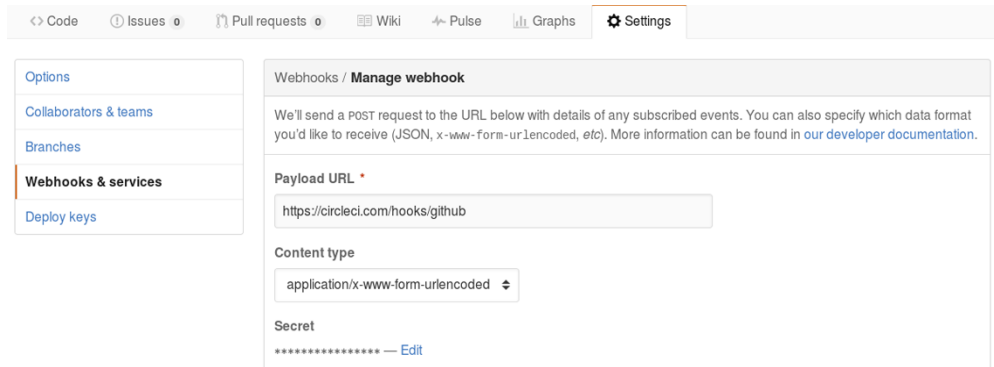
Code repositories and CI platforms like GitHub and CircleCI implement a concept called *webhooks* to pass notifications around. When a change happens in the code repository, a webhook pushes a notification to a web address hosted by the CI platform. The body of the notification contains information about the change the CI platform uses to perform tasks.

When you sign in to CircleCI using your GitHub account, CircleCI asks you for permission to perform actions on your behalf in your GitHub account. One of these actions will be to automatically configure a webhook into the invoicer's GitHub repository to notify CircleCI of new events. Figure 2.2 shows the result of the automatic webhook configuration in GitHub.

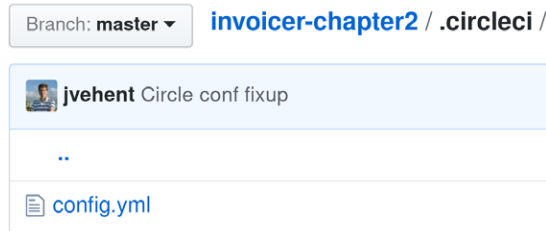
This webhook is used in steps 2 and 4 of figure 2.1. Every time GitHub needs to notify CircleCI of a change, GitHub posts a notification to <https://circleci.com/hooks/github>. CircleCI receives the notification and triggers a build at the invoicer. The simplicity of the webhook technique makes it popular for interface services operated by different entities.

### **Security note**

GitHub has a sophisticated permission model allowing users to delegate fine-grained permissions to third-party applications. Yet, CI platforms want read and write access to all the repositories of a user. Rather than using your own highly privileged user to integrate with a CI platform, in chapter 6 we'll discuss how to use a low-privilege account and keep your accesses under control.



**Figure 2.2** The webhook between GitHub and CircleCI is automatically created in the invoicer's repository to trigger a build of the software when changes are applied.



**Figure 2.3** The CircleCI configuration is stored under the .circleci directory in the repository of the application.

The config.yml file shown in figure 2.3 is placed in the repository of the application. It is written in YAML format and configures the CI environment to run specific tasks on every change recorded by GitHub. Specifically, you'll configure CircleCI to test and compile the invoicer application, and then build and publish a Docker container, which you'll later deploy to the AWS environment.

**NOTE** YAML is a data-serialization language commonly used to configure applications. Compared to formats like JSON or XML, YAML has the benefit of being much more accessible to humans.

The full CircleCI configuration file is shown next. You may notice some parts of the file are command-line operations, whereas others are parameters specific to CircleCI. Most CI platforms allow operators to specify command-line operations, which makes them well suited to run custom tasks.

#### Listing 2.1 config.yml configures CircleCI for the application

```
version: 2
jobs:
  build:
    working_directory:
      ➡ /go/src/github.com/Securing-DevOps/invoicer-chapter2
```

Configures a working directory to build the Docker container of the application



```

docker:
- image: circleci/golang:1.8
steps:
- checkout
- setup_remote_docker

- run:
  name: Setup environment
  command: |
    gb="/src/github.com/${CIRCLE_PROJECT_USERNAME}";
    if [ ${CIRCLE_PROJECT_USERNAME} == 'Securing-DevOps' ]; then
      dr="securingdevops"
    else
      dr=${DOCKER_USER}
    fi
    cat >> $BASH_ENV << EOF
    export GOPATH_HEAD="$(echo ${GOPATH}|cut -d ':' -f 1)"
    export GOPATH_BASE="$(echo ${GOPATH}|cut -d ':' -f 1)${gb}"
    export DOCKER_REPO="${dr}"
    EOF

- run: mkdir -p "${GOPATH_BASE}"
- run: mkdir -p "${GOPATH_HEAD}/bin"

- run:
  name: Testing application
  command: |
    go test \
    github.com/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPONAME}

- deploy:
  command: |
    if [ "${CIRCLE_BRANCH}" == "master" ]; then
      docker login -u ${DOCKER_USER} -p ${DOCKER_PASS};
      go install --ldflags '-extldflags "-static"' \
      github.com/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPONAME};
      mkdir bin;
      cp "$GOPATH_HEAD/bin/${CIRCLE_PROJECT_REPONAME}" bin/invoicer;
      docker build -t ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME} .;
      docker images --no-trunc | awk '/^app/ {print $3}' | \
      sudo tee ${CIRCLE_ARTIFACTS}/docker-image-shasum256.txt;
      docker push ${DOCKER_REPO}/${CIRCLE_PROJECT_REPONAME};
    fi

```

**Declares the environment the job will run on**

**Environment variables needed to build the application**

**Runs the unit tests of the application**

**If changes are applied to the master branch, builds the Docker container of the application**

**Builds the application binary**

**Logs into the Docker Hub service**

**Builds a container of the application using a Dockerfile**

**Pushes the container to Docker Hub**

Parts of this file may appear obscure, particularly Docker and Go. Ignore them for now; we'll get back to them later, and focus on the idea behind the configuration file. As you can see in this listing, the syntax is declarative, similar to how we'd write a shell script that performs these exact operations.

The configuration file must be kept in the code repository. When present, CircleCI will use its instructions to take actions when a webhook notification is received from GitHub. To trigger a first run, add the configuration file from listing 2.1 to a feature branch of the Git repository, and push the branch to GitHub.

### Listing 2.2 Creating a Git feature branch with a patch to add the CircleCI configuration

Creates a Git feature branch

```
$ git checkout -b featbr1
$ git add .circleci/config.yml
$ git commit -m "initial circleci conf"
$ git push origin featbr1
```

Adds config.yml to the branch

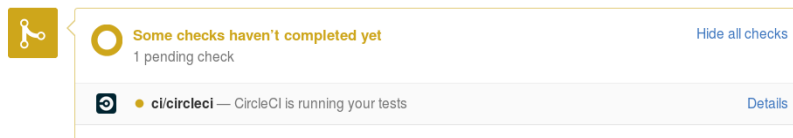
Pushes changes to the code repository

For CircleCI to run the tests defined in config.yml, create a pull request to merge the patch from the feature branch into the master branch.

### What is a pull request?

“Pull request” is a term popularized by GitHub that represents a request to pull changes from a given branch into another branch, typically between a feature and a master branch. A pull request is opened when a developer submits a patch for review. Webhooks triggers on pull requests to run automated tests in CI (see step 2 of figure 2.1), and peers review the proposed patch before agreeing to merge it (see step 3 of figure 2.1).

Figure 2.4 shows the user interface of a GitHub pull request waiting for tests in CircleCI to finish. CircleCI retrieves a copy of the feature branch, reads the configuration in config.yml and follows all the steps to build and test the application.



**Figure 2.4** The web interface of a GitHub pull request displays the status of tests running in CircleCI. Running tests are yellow; they turn green if CircleCI completed successfully, or red if a failure was encountered.

Note that, per your configuration, only unit tests that run as part of the `go test` command are executed. The `deploy` section of the configuration will only be executed after the pull request is accepted and code is merged into the master branch.

Let’s assume that your reviewer is satisfied with the changes and approves the pull request, completing step 3 of the pipeline. The patch is merged into the master branch and the pipeline enters steps 4 and 5 of figure 2.1. CircleCI will run again, execute the deployment section to build a Docker container of the application, and push it to Docker Hub.

## 2.4 The container repository: Docker Hub

Our CircleCI configuration shows several commands that call Docker to build a container for the application, such as `docker build` and `docker push`. In this section, I first explain why Docker is an important component of DevOps, and then we'll take a close look at how the container is built.

Containers, and Docker containers in particular, are popular because they help solve the complex problem of managing code dependencies. Applications usually rely on external libraries and packages to avoid reimplementing common code. On systems, operators prefer to share these libraries and packages for ease of maintenance. If an issue is found in one library used by 10 applications, only that one library is updated, and all applications automatically benefit from the update.

Issues arise when various applications require different versions of the same library. For example, a package wanting to use OpenSSL 1.2 on a system that uses OpenSSL 0.9 by default won't work. Should the base system have all versions of OpenSSL installed? Are they going to conflict? The answer is rarely simple, and these issues have caused many headaches for operators and developers. This problem has several solutions, all of which are based on the idea that applications should manage their dependencies in isolation. Containers provide a packaging mechanism to implement this kind of isolation.

### New to Docker?

In this chapter, we focus on a limited usage of Docker containers to package the *invoicer* application. For a full introduction to Docker, please refer to Jeff Nickoloff's *Docker in Action* (Manning, 2016).

As shown in the CircleCI configuration file we discussed previously, Docker containers are built according to a configuration file called a Dockerfile. Docker does a good job of abstracting the tedious task of building, shipping, and running containers. The Dockerfile that follows is used to build the container of the *invoicer* application. It's short, yet hides a surprising amount of complexity. Let's examine what it does.

### Listing 2.3 Dockerfile used to build the invoicer's container

```
FROM busybox:latest
RUN addgroup -g 10001 app && \
    adduser -G app -u 10001 \
    -D -h /app -s /sbin/nologin app
COPY bin/invoicer /bin/invoicer
USER app
EXPOSE 8080
ENTRYPOINT /bin/invoicer
```

Let's examine listing 2.3:

- The `FROM` directive indicates a base container used to build your own container. Docker containers have *layers* which allow you to add information on top of another container. Here, we use a container based on BusyBox, a minimal set of common Linux tools.

- The RUN directive creates a user called “app” which is then used by the USER directive to execute your application.
- The COPY command loads the executable of the invoicer on the container. This command takes the local file from bin/invoicer (a path relative to where the build operation runs) and puts it into /bin/invoicer in the container.
- EXPOSE and ENTRYPOINT run the invoicer application when the container starts and allow outsiders to talk to its port, 8080.

To build a container with this configuration, first compile the source code of the *invoicer* into a static binary, copy it into bin/invoicer, then use `docker build` to create the container.

#### Listing 2.4 Compiling the invoicer into a static binary

```
go install --ldflags '-extldflags "-static"' \  
  github.com/Securing-DevOps/invoicer-chapter2 \  
cp "$GOPATH/bin/invoicer-chapter2" bin/invoicer
```

Packaging the invoicer binary into a Docker container is then done via the build command.

#### Listing 2.5 Creating the invoicer container via the docker build command

```
docker build -t securingdevops/invoicer-chapter2 -f Dockerfile .
```

That’s all you need for Docker to build your application container. CircleCI will run this exact command and follow with a push of the container to Docker Hub.

Pushing to Docker Hub requires an account on <https://hub.docker.com/> and a repository called “securingdevops/invoicer” (or any other name that matches your GitHub username and repository name). CircleCI needs these account credentials to log into Docker Hub, so after creating the account, head over to the Settings section of the repository in CircleCI to set the DOCKER\_USER and DOCKER\_PASS environment variables to the username and password of Docker Hub.

#### Security notes

You should avoid sharing your own Docker Hub credentials with CircleCI. In chapter 6, we’ll discuss how service-specific accounts with minimal privileges can be used for this purpose.

Most CI platforms support mechanisms to use sensitive information without leaking secrets. Both CircleCI and Travis CI protect environment variables that contain secrets by refusing to expose them to pull requests coming from outside the repository (forks instead of feature branches).

Let’s summarize what you’ve implemented so far. You have a source-code repository that calls a CI platform using webhooks when changes are proposed. Tests run automatically to help reviewers verify that the changes don’t break functionalities. When a change is approved, it’s merged into a master branch. The CI platform is then invoked

a second time to build a container of the application. The container is uploaded to a remote repository where everyone can retrieve it.

### **In-house CI**

You can achieve exactly the same results using a pipeline operated entirely behind closed doors. Replace GitHub with a private instance of GitLab, replace CircleCI with Jenkins, and run your own Docker Registry server to store containers, and the same workflow will be implemented on a private infrastructure (but will take much longer to set up).

The core concept of the CI pipeline remains regardless of how you implement it. Automate the testing and building steps that happen at every change of the application, to accelerate the integration of changes while guaranteeing stability.

The CI pipeline completely automates testing and packaging the invoicer application. It can run hundreds of times a day if needed, and will reliably transform code into an application container you can ship to production. The next phase is to build an infrastructure to host and run that container.

## **2.5 The production infrastructure: Amazon Web Services**

Back in college, my law professor used to tell the story of what was probably the first web-hosting service operated in France. It was run by a friend of his in the early 1990s. At the time, hosting a web page on the newly born internet required operating everything, from the network to the system layers. My professor's friend didn't have the means to pay for a data center, so he laid out stacks of hard drives, motherboards, and cables on desks in his basement and maintained connectivity to the internet through a handful of modems modified for this purpose. The result was a noisy monster of spinning and scratching disks, and probably a huge fire hazard, but it worked and hosted websites!

The origins of the web are full of similar stories. They now serve to highlight the progress we made in building and operating online services. Up until the late 2000s, building an entire infrastructure from the ground up was a complicated and tedious task that required lots of hardware and wiring. Nowadays, most organization outsource this complexity to specialized companies, and focus their energy on building their core products.

IaaS providers have simplified the task of building infrastructure by handling the complexity in the background and only exposing simple interfaces to operators. Heroku, Google Cloud, Microsoft Azure, Cloud Foundry, Amazon Web Services, and IBM Cloud are examples from the long list of providers that will manage the infrastructure for you. IaaS users only need to declare the infrastructure at a logical level and let the provider translate the declaration to the physical layer. Once declared, the operator will entirely manage the infrastructure. By the time you're done with the initial setup, the management of the invoicer will be outsourced to the provider, and you won't be managing infrastructure components at all.

In this section, we focus on AWS, and more specifically on its Elastic Beanstalk (EB) service. EB is specifically designed to host containers and abstract the management of the infrastructure away from the operator. The choice of using EB for the purpose of this book is completely arbitrary. It doesn't have any distinctive features, other than being simple enough to manage to fit within this chapter and demonstrate how to implement a cloud service in AWS.

Before we get to the technical bits, we first need to discuss the concept of three-tier architecture, which you'll implement to host the invoicer. Next, we'll go through a step-by-step deployment of the invoicer in AWS EB.

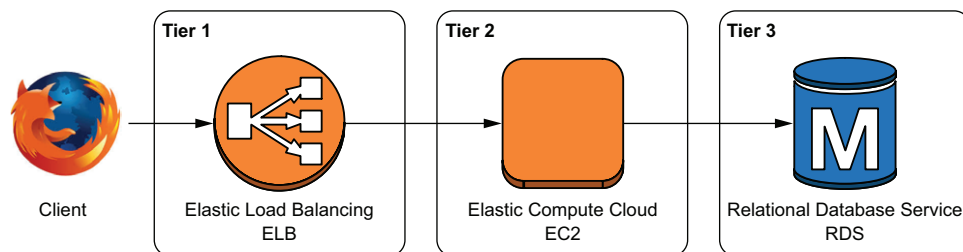
### New to Amazon Web Services?

From here on, I assume the reader has been introduced to AWS and can perform basic tasks in the platform. For the reader who is new to AWS, an excellent introduction can be found in Michael Wittig and Andreas Wittig's *Amazon Web Services in Action* (Manning, 2015). The infrastructure presented here can be run in the free tier of AWS, so you can experiment for free with your own account.

## 2.5.1 Three-tier architecture

A common pattern in web applications is the three-tier architecture represented in figure 2.5:

- The first tier handles incoming HTTP requests from clients (web browsers or client applications). Caching and load balancing can be performed at this level.
- The second tier processes requests and builds responses. This is typically where the core of the application lives.
- The third tier is the database and other backends that store data for the application.



**Figure 2.5** A three-tier architecture in AWS shows a load-balancer layer (tier 1), followed by a compute node (tier 2), and backed by a relational database (tier 3).

Figure 2.5 uses the official AWS terminology and icons. We'll reuse them throughout the book, so it's best to familiarize yourself with their roles right away.

- *ELB*—Elastic Load Balancing is an AWS-managed service that receives traffic from internet clients and distributes it to applications. The main goal of ELB is to allow applications to augment and reduce the number of servers as needed without touching the frontend of the service. ELB also provides SSL/TLS termination to handle HTTPS in applications easily.
- *EC2*—An Elastic Compute Cloud instance is nothing more than a virtual machine (VM) that runs an operating system (OS). The base infrastructure of EC2 is managed by AWS, and only the system on the VM—not the hypervisor or network underneath it—is accessible to the operator. You’ll run applications on EC2 instances.
- *RDS*—Most applications need to store data and thus need a database. Relational Database Service (RDS) provides MySQL, PostgreSQL, and Oracle databases managed entirely by AWS, allowing the DevOps team to focus on the data and not management of the database servers. In the example, we use PostgreSQL to store the invoicer’s data.

Online services are often more complex than the example in figure 2.5, but their architecture is almost always based on the three-tier approach. The invoicer is a three-tier application as well. In the next section, I explain how to create this environment in AWS using the Elastic Beanstalk (EB) service.

## 2.5.2 *Configuring access to AWS*

You’ll use the official AWS command-line tool to create the AWS EB infrastructure, which needs a little bit of setup. First, retrieve access credentials for your account from the Identity and Access Management (IAM) section of the web console. On your local machine, access keys should be stored in `$HOME/.aws/credentials`. You can organize multiple access keys per profile, but for now limit yourself to one access key in the default profile, as shown in the next listing.

### Listing 2.6 AWS credentials in `$HOME/.aws/credentials`

```
[default]
aws_access_key_id = AKIAILJA79QHF28ANU3
aws_secret_access_key = iqdoh181HoqOQ08165451dNui18Oah8913Ao8HTn
```

You also need to tell AWS which region you prefer to use by declaring it in `$HOME/.aws/config`. We’ll work in the US East 1 region, but you could also pick a region closer to where the target users are to reduce network latency.

### Listing 2.7 AWS default region configuration in `$HOME/.aws/config`

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

The standard tools AWS provides know to look for configuration in these locations automatically. Install one of the most popular tools, `awscli`, that provides the “aws” command line. It’s a Python package installable via `pip` (or Homebrew on macOS only).

**Listing 2.8 Installing awscli tools via pip**

```
$ sudo pip install -U awscli  
  
Successfully installed awscli-1.10.32
```

**Package managers**

Pip and Homebrew are package managers. Pip is the standard Python package manager that works on all operating systems. Homebrew is a package manager specific to macOS, managed by a community of contributors.

Although the installation package is called *awscli*, the command it provides is called *aws*. The aws command line is a powerful tool that can control an entire infrastructure. You'll spend a lot of time with it and gradually familiarize yourself with the various commands.

**Creation EB script**

The aws commands used in the rest of this chapter to create the Elastic Beanstalk environment have been bundled into a shell script available at [https://securing-devops.com/eb\\_creation\\_script](https://securing-devops.com/eb_creation_script). Feel free to use it if entering commands manually isn't your thing.

### 2.5.3 Virtual Private Cloud

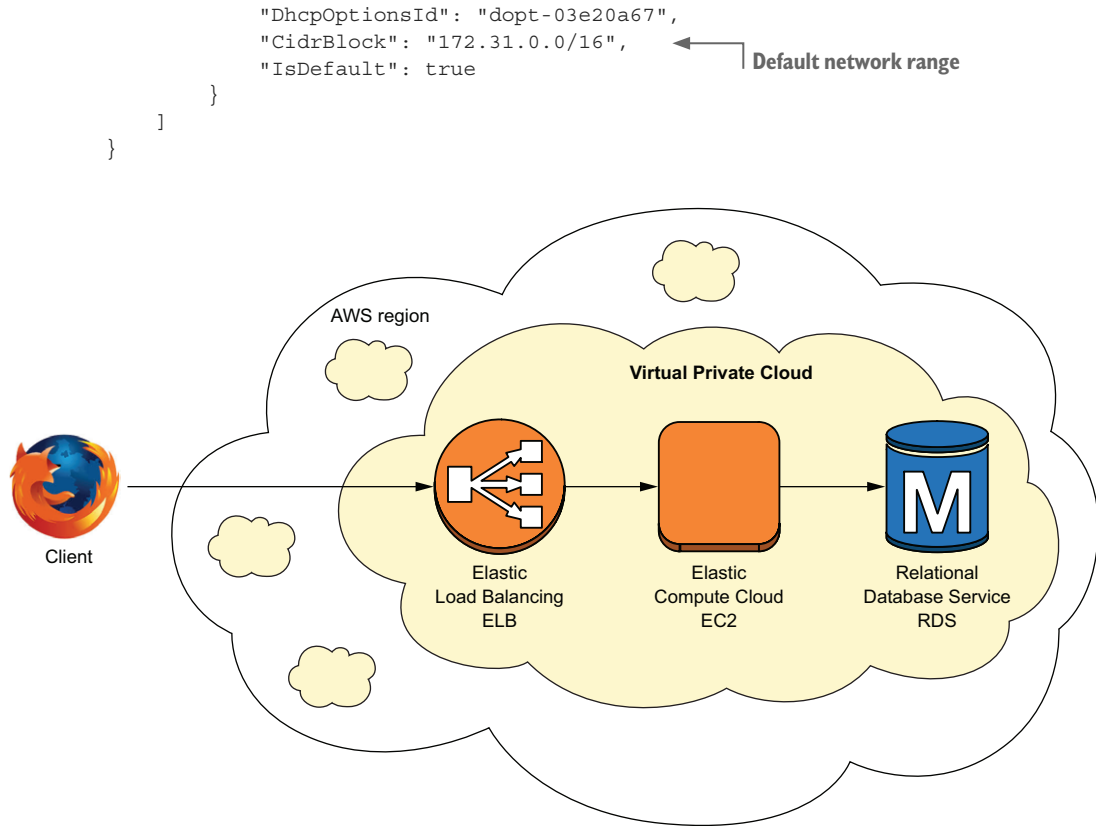
All AWS accounts come with a Virtual Private Cloud (VPC) assigned by default to the account in each region. As shown in figure 2.6, a VPC is a segment of the AWS network dedicated to a customer within the infrastructure of a given region. VPCs are isolated from each other and have networking capabilities we'll use later. At a physical level, all customers share the same networking equipment, but that view is entirely abstracted away by the IaaS.

You can retrieve the ID of the VPC created with your account in the us-east-1 region using the AWS command line in the next listing.

**Listing 2.9 Retrieving the unique ID of the VPC using the AWS command line**

```
$ aws ec2 describe-vpcs  ← Calls the API to retrieve VPC details  
{  
  "Vpcs": [  
    {  
      "VpcId": "vpc-2817dc4f", ← VPC unique ID  
      "InstanceTenancy": "default",  
      "State": "available",
```





**Figure 2.6** Each internal cloud represents a VPC and is private to a specific customer of AWS. By default, VPCs can't talk to each other and provide a virtual isolation layer between customers.

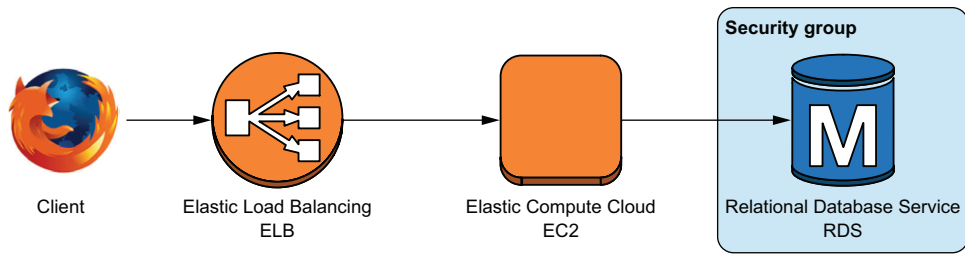
The command returns the `vpc-2817dc4f` ID for the default VPC. This ID is unique and will be different when you set up your own account. Each AWS account can have several VPCs to host components, but for our purposes, using the default VPC will be fine.

### 2.5.4 Creating the database tier

The next step of the setup is to create the third tier of your infrastructure: the database, as shown in figure 2.7. This tier is composed of an RDS instance running PostgreSQL placed into a security group. You need to define the security group first, and then place the instance into it.

#### What are security groups?

Security groups are virtual domains that control interactions between AWS components. We'll discuss security groups further in chapter 4 when covering infrastructure security.



**Figure 2.7** The third tier of the invoicer infrastructure is made of an RDS inside its security group.

Creating a security group with the AWS command line is done using the following parameters. For now, the security group doesn't allow or deny anything; it's only declared for future use.

#### Listing 2.10 Creating the security group of the RDS instance

```
$ aws ec2 create-security-group \
--group-name invoicer_db \
--description "Invoicer database security group" \
--vpc-id vpc-2817dc4f \
{
  "GroupId": "sg-3edf7345"
}
```

Unique name of the security group

ID of the default VPC

Response from the API with the unique security group ID

Next, create the database and place it inside the sg-3edf7345 security group.

#### Listing 2.11 Creating the RDS instance

```
$ aws rds create-db-instance \
--db-name invoicer \
--db-instance-identifier invoicer-db \
--vpc-security-group-ids sg-3edf7345 \
--allocated-storage "5" \
--db-instance-class "db.t2.micro" \
--engine postgres \
--engine-version 9.6.2 \
--auto-minor-version-upgrade \
--publicly-accessible \
--master-username invoicer \
--master-user-password 'S0m3th1ngr4nd0m' \
--no-multi-az
```

Name of the RDS instance ID

ID of the security group

Configuration of the PostgreSQL instance

Admin credentials of the database

Listing 2.11 has a lot packed into it. AWS creates a VM designed to run PostgreSQL 9.5.2. The VM has minimal resources (low CPU, memory, network throughput, and disk space), as determined by the allocated storage of 5 GB and the db.t2.micro instance class. Finally, AWS creates a database inside PostgreSQL called “invoicer” and

grants administrator permissions to a user also called “invoicer” with the password “\$0m3th1ngr4nd0m.”

The creation of an RDS instance can take some time, as AWS needs to find an appropriate location for it across its physical infrastructure and run through all the configuration steps. You can monitor the creation of the instance with the `describe-db-instances` flag of the AWS command line, as shown in the following listing. The script monitors the AWS API every 10 seconds and exits the loop when a host name for the database is returned in the JSON response.

#### Listing 2.12 Monitoring loops that wait for the RDS instance to be created

```
while true; do
  aws rds describe-db-instances \
    --db-instance-identifier invoicer-db > /tmp/invoicer-db.json
  dbhost=$(jq -r '.DBInstances[0].Endpoint.Address' /tmp/invoicer-db.json)
  if [ "$dbhost" != "null" ]; then break; fi
  echo -n '.'
  sleep 10
done
echo "dbhost=$dbhost"

...dbhost=invoicer-db.cxuqrkdqhk1f.us-east-1.rds.amazonaws.com
```

#### Querying JSON with jq

Note the use of the `jq` utility to parse the JSON response from the AWS API. `Jq` is a popular command-line tool to extract information from JSON-formatted data without involving a programming language. You can learn more about it at <https://stedolan.github.io/jq/>. On Ubuntu, install it with `apt-get install jq`. On macOS, `brew install jq` will work.

Once created, your database instance will have a hostname internal to the VPC and gated by a security group. You’re ready to create the first and second tiers of the infrastructure.

### 2.5.5 *Creating the first two tiers with Elastic Beanstalk*

AWS provides many different techniques to deploy applications and manage servers. In this example, we use what’s probably the most automated of them: Elastic Beanstalk (EB). EB is a management layer on top of other AWS resources. It can be used to create ELBs and EC2 instances and their security groups, and to deploy applications to them. For this example, deploy the Docker container you built in the CI pipeline to EC2 instances fronted by an ELB and managed by EB. The architecture is shown in figure 2.8.

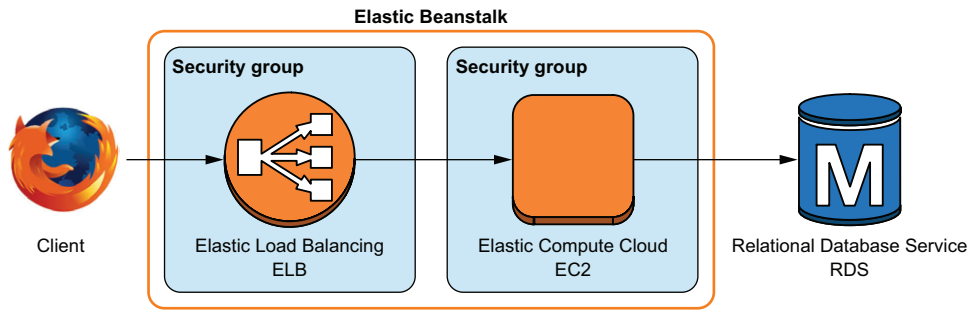


Figure 2.8: The first and second tiers of the infrastructure are managed by AWS EB.

EB first needs an “application,” which is an empty structure to organize your components. Create one for the invoicer with the following command.

#### Listing 2.13 Creating an EB application

```
aws elasticbeanstalk create-application \
  --application-name invoicer \
  --description "Securing DevOps Invoicer application"
```

Inside the invoicer EB application, create an environment that will run the invoicer’s Docker container. This part of the configuration requires more parameters, because you need to indicate which solution stack you want to use. Solution stacks are pre-configured EC2 instances for a particular use case. We want to use the latest version preconfigured to run the Docker instance. You can obtain its name using the `list-available-solution-stacks` command, and filter its output using `jq` and `grep`.

#### Listing 2.14 Retrieving the name of the latest Docker EB stack available

```
aws elasticbeanstalk list-available-solution-stacks | \
  jq -r '.SolutionStacks[]' | \
  grep -P '.*Amazon Linux.+Docker.+ ' | \
  head -1
```

Extracts fields from the JSON response

64bit Amazon Linux 2017.03 v2.7.3 running Docker 17.03.1-ce

### What about performances?

You may notice we run a Docker container inside a VM that runs on top of a hypervisor. This may seem rather inefficient. It’s true that the raw performance of this approach is lower than running applications on bare-metal servers, but the ease of deployment and maintenance—which lets us easily increase the number of servers with the load—mostly offsets the performance hit. It all comes down to what matters the most to you: raw performance or deployment flexibility.

The version of this Docker solution stack will likely have changed by the time you read these pages, but you can always use the AWS API to obtain the name of the latest version.

Before you create the environment, you need to prepare the configuration of the invoicer application. Every application needs configuration parameters typically provided in configuration files on the filesystem of the application servers. Creating and updating those files, however, requires direct access to servers, which you want to avoid here.

If you have a look at the source code of the invoicer, you'll notice that the only configuration it needs is the parameters to connect to its PostgreSQL database. Rather than managing a configuration file, those parameters can be taken from the environment variables. The following listing shows how the invoicer reads its database configuration from four environment variables.

#### Listing 2.15 Go code to get PostgreSQL parameters from environment variables

```
db, err = gorm.Open("postgres",
    fmt.Sprintf("postgres://%s:%s@%s/%s?sslmode=%s",
        os.Getenv("INVOICER_POSTGRES_USER"),
        os.Getenv("INVOICER_POSTGRES_PASSWORD"),
        os.Getenv("INVOICER_POSTGRES_HOST"),
        os.Getenv("INVOICER_POSTGRES_DB"),
        "disable",
    ))
if err != nil {
    panic("failed to connect database")
}
```

Retrieves configuration from environment variables

Upon startup, the invoicer will read the four environment variables defined in listing 2.15 and use them to connect to the database. You need to configure those variables in EB so they can be passed to the application, through Docker, at startup. This is done in a JSON file, shown next, loaded in the environment creation command. The content of the following listing is saved in a text file named `ebs-options.json`.

#### Listing 2.16 `ebs-options.json` references variables used to connect to the database

```
[
  {
    "Namespace": "aws:elasticbeanstalk:application:environment",
    "OptionName": "INVOICER_POSTGRES_USER",
    "Value": "invoicer"
  },
  {
    "Namespace": "aws:elasticbeanstalk:application:environment",
    "OptionName": "INVOICER_POSTGRES_PASSWORD",
    "Value": "S0m3th1ngr4nd0m"
  },
  {
    "Namespace": "aws:elasticbeanstalk:application:environment",
    "OptionName": "INVOICER_POSTGRES_DB",
    "Value": "invoicer"
  }
],
```

```

{
  "Namespace": "aws:elasticbeanstalk:application:environment",
  "OptionName": "INVOICER_POSTGRES_HOST",
  "Value": "invoicer-db.cxuqrkdqhk1f.us-east-1.rds.amazonaws.com"
}
]

```

### Security note

Instead of using the database administrator account in your application, you should create a separate user that has limited database permissions. We'll discuss how database permissions can be used to protect against application compromises in chapter 4.

Save the file under the name `ebs-options.json`, and proceed with the creation of the environment.

#### Listing 2.17 Creating the EB environment to run the application container

```

aws elasticbeanstalk create-environment \
  --application-name invoicer \
  --environment-name invoicer-api \
  --description "Invoicer APP" \
  --solution-stack-name \
    "64bit Amazon Linux 2017.03 v2.7.3 running Docker 17.03.1-ce" \
  --option-settings file://$(pwd)/ebs-options.json \
  --tier "Name=WebServer,Type=Standard,Version='"

```

Application name created previously

EB takes care of the creation of the EC2 instances and ELB of the environment, creating the first two tiers of the infrastructure in a single step. This step will take several minutes to complete, because various components need to be instantiated for the first time. Once finished, the public endpoint to access the application can be retrieved using the `describe-environments` command.

#### Listing 2.18 Retrieving the public hostname of the EB load balancer

```

aws elasticbeanstalk describe-environments \
  --environment-names invoicer-api \
  | jq -r '.Environments[0].CNAME'

```

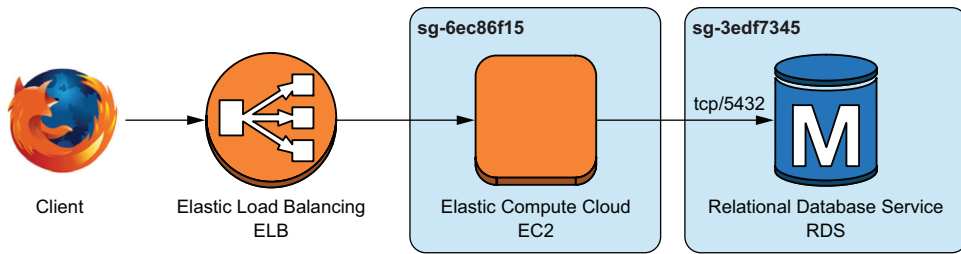
Public endpoint

invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com

### Security note

EB creates an ELB that only supports HTTP, not HTTPS. Configuring an ELB to support HTTPS, including which SSL/TLS configuration to use, is explained in chapter 5.

Your environment is set up, but the EC2 instance isn't yet permitted to connect to the database. Security groups block all inbound connectivity by default, so you need to open the security group of the RDS instance to allow the EC2 instance to connect, as shown in figure 2.9.



**Figure 2.9** The security group of the RDS instance must permit inbound connections to allow the EC2 instance to reach the database.

You already know the ID of the RDS security group is sg-3edf7345. You need to insert a rule into it that permits everyone, aka 0.0.0.0/0, to connect to it.

#### Listing 2.19 Opening the RDS security group to all origins

```
aws ec2 authorize-security-group-ingress \
--group-id sg-3edf7345 \
--cidr 0.0.0.0/0 \
--protocol tcp --port 5432
```

Application name created previously

Opens up to the whole internet

Permits PostgreSQL port

#### Security note

You can certainly do better than opening up your database to the whole internet. In chapter 4, we'll discuss how to use security groups to manage dynamic and fine-grained firewall rules.

At this point of the setup, you have a fully operational infrastructure, but nothing running on it yet. The next phase is to deploy the Docker container of the invoicer, which you built and published previously, to your EB infrastructure.

### 2.5.6 Deploying the container onto your systems

The Docker container of the invoicer is hosted on [hub.docker.com](https://hub.docker.com) (step 5 of figure 2.1). You need to tell EB the location of the container so it can pull it down from Docker Hub and deploy it to the EC2 instance. The following JSON file will handle that declaration.

#### Listing 2.20 EB configuration indicates the location of the container

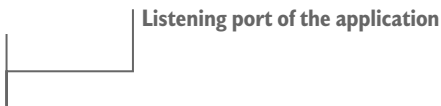
```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "docker.io/securingdevops/invoicer",
    "Update": "true"
  },
}
```

Location of the invoicer container on Docker Hub

```

"Ports": [
  {
    "ContainerPort": "8080"
  }
],
"Logging": "/var/log/nginx"
}

```




The JSON configuration will be read by each new instance that joins your EB infrastructure, so you need to make sure instances can retrieve the configuration by uploading it to AWS S3. Save the definition to a local file, and upload it using the command line. Make sure to change the bucket name from “invoicer-eb” to something personal, as S3 bucket names must be unique across all AWS accounts.

#### Listing 2.21 Uploading the application configuration to S3

```

aws s3 mb s3://invoicer-eb
aws s3 cp app-version.json s3://invoicer-eb/

```



In EB, you reference the location of the application definition to create an application version named `invoicer-api`.

#### Listing 2.22 Assigning the application configuration to the EB environment

```

aws elasticbeanstalk create-application-version \
  --application-name "invoicer" \
  --version-label invoicer-api \
  --source-bundle "S3Bucket=invoicer-eb,S3Key=app-version.json"

```

And finally, instruct EB to update the environment using the `invoicer-api` application version you just created. With one command, tell AWS EB to pull the Docker image, place it on the EC2 instances, and run it with the environment previously configured, all in one automated step. Moving forward, the command in the following listing is the only one you’ll need to run to deploy new versions of the application.

#### Listing 2.23 Deploying the application configuration to the EB environment

```

aws elasticbeanstalk update-environment \
  --application-name invoicer \
  --environment-id e-curu6awket \
  --version-label invoicer-api

```

The environment update takes several minutes, and you can monitor completion in the web console. When the environment turns green, it’s been updated successfully. The `invoicer` has a special endpoint on `/__version__` that returns the version of the application currently running. You can test the deployment by querying the version endpoint from the command line and verifying the version returned is the one you expect.



**Listing 2.24 Retrieving the application version through its public endpoint**

```
curl \
http://invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com/__version__
{
  "source": "https://github.com/Securing-DevOps/invoicer",
  "version": "20160522.0-660c2c1",
  "commit": "660c2c1bcece48115b3070ca881b1a7f1c432ba7",
  "build": "https://circleci.com/gh/Securing-DevOps/invoicer/"
}
```

Make sure the database connection works as expected by creating and retrieving an invoice.

**Listing 2.25 Creating an invoice via the public API**

```
curl -X POST \
--data '{"is_paid": false, "amount": 1664, "due_date":
      "2016-05-07T23:00:00Z", "charges": [ { "type": "blood work", "amount":
      1664, "description": "blood work" } ] }' \
http://invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com/invoice

created invoice 1
```

Your first invoice was successfully created. That's encouraging. Now let's try to retrieve it.

**Listing 2.26 Retrieving an invoice via the public API**

```
curl \
http://invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com/invoice/1

{
  "ID": 1,
  "CreatedAt": "2016-05-25T18:49:04.978995Z",
  "UpdatedAt": "2016-05-25T18:49:04.978995Z",
  "amount": 1664,
  "charges": [
    {
      "ID": 1,
      "CreatedAt": "2016-05-25T18:49:05.136358Z",
      "UpdatedAt": "2016-05-25T18:49:05.136358Z",
      "amount": 1664,
      "description": "blood work",
      "invoice_id": 1,
      "type": "blood work"
    }
  ],
  "due_date": "2016-05-07T23:00:00Z",
  "is_paid": false,
  "payment_date": "0001-01-01T00:00:00Z"
}
```

**Security note**

An invoice-management API left wide open to the internet is obviously a bad idea. In chapter 3, we'll discuss how to protect web applications, using authentication.

This is it: the invoicer is up and running in AWS Elastic Beanstalk. Getting to this point took a significant amount of work, but look at what you achieved: with one command, you can now deploy new versions of the invoicer. No server management, no manual configuration, everything from testing the code, to deploying the container, to production is automated. You can go from the patch sent to the source code repository to deployment in the infrastructure well within the 15 minutes we decided on at the beginning of the chapter.

Our infrastructure is still naive and doesn't have all the security controls required to operate a production service. But that's configuration. The logic behind the CI/CD pipeline will remain unchanged as we bring more security to the infrastructure. We'll maintain the capability to deploy new versions of applications without involving manual steps, all within the 15-minute window.

That's the promise of DevOps: fully automated environments that allow the organization to go from idea to product in short cycles. With less pressure on the operational side, the organization is free to focus on its product more, including its security.

## 2.6 A rapid security audit

As we focused on getting the invoicer deployed, we ignored several security issues on the application, infrastructure, and CI/CD pipeline:

- GitHub, CircleCI, and Docker Hub need access to each other. By default, we granted all three access to highly privileged accounts which, if leaked, could damage other services hosted on these accounts. Making use of accounts with fewer privileges will increase security.
- Similarly, the credentials we used to access AWS could easily be leaked, granting a bad actor full access to the environment. Multifactor authentication and fine-grained permissions should be used to reduce the impact of a credential leak.
- Our database security practices are subpar. Not only does the invoicer use an admin account to access PostgreSQL, but the database itself is also public. A good way to reduce the risk of a breach is to harden the security of the database.
- The public interface to the invoicer uses clear-text HTTP, meaning that anyone on the connection path can copy and modify the data in transit. HTTPS is an easy security win and we should make use of it right away.
- And finally, the invoicer itself is wide open to the internet. We need authentication and strong security practices to keep the application secure.

Throughout the rest of part 1, we'll work through these issues and discuss how to add security. We've got some work to do, and four chapters to secure your DevOps pipeline:

- We'll start with application security in chapter 3 and discuss vulnerabilities and controls the invoicer is exposed to.
- Infrastructure security will be discussed in chapter 4 where we harden the AWS environment that hosts the production service.
- Guaranteeing communications security with the invoicer will be done in chapter 5 when we implement HTTPS.
- Pipeline security is the topic of chapter 6 and will cover the security principles of building and deploying code in CI/CD.

### **Summary**

- Continuous integration interfaces components via webhooks to test code and build containers.
- Continuous delivery uses IaaS, like AWS Elastic Beanstalk, to deploy containers to production.
- Except for manual reviews, all steps of the CI/CD pipeline are fully automated.
- A barebones DevOps pipeline is riddled with security problems.

# Securing DevOps

Julien Vehent



An application running in the cloud can benefit from incredible efficiencies, but they come with unique security threats too. A DevOps team's highest priority is understanding those risks and hardening the system against them.

**Securing DevOps** teaches you the essential techniques to secure your cloud services. Using compelling case studies, it shows you how to build security into automated testing, continuous delivery, and other core DevOps processes. This experience-rich book is filled with mission-critical strategies to protect web applications against attacks, deter fraud attempts, and make your services safer when operating at scale. You'll also learn to identify, assess, and secure the unique vulnerabilities posed by cloud deployments and automation tools commonly used in modern infrastructures.

## What's Inside

- An approach to continuous security
- Implementing test-driven security in DevOps
- Security techniques for cloud services
- Watching for fraud and responding to incidents
- Security testing and risk assessment

Readers should be comfortable with Linux and standard DevOps practices like CI, CD, and unit testing.

**Julien Vehent** is a security architect and DevOps advocate. He leads the Firefox Operations Security team at Mozilla, and is responsible for the security of Firefox's high-traffic cloud services and public websites.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [www.manning.com/books/securing-devops](http://www.manning.com/books/securing-devops)

“Provides both sound ideas and real-world examples. A must-read.”

—Adrien Saladin, PeopleDoc

“Makes a complex topic completely approachable. Recommended for DevOps personnel and technology managers alike.”

—Adam Montville  
Center for Internet Security

“Practical and ready for immediate application.”

—Yan Guo, Eventbrite

“An amazing resource for secure software development—a must in this day and age—whether or not you're in DevOps.”

—Andrew Bovill, Next Century

ISBN-13: 978-1-61729-413-6  
ISBN-10: 1-61729-413-6



9 781617 294136



\$49.99 / Can \$65.99 [INCLUDING eBook]