# 10 Microservice APIs in Kubernetes

This chapter covers

- Deploying an API to Kubernetes
- Hardening Docker container images
- Setting up a service mesh for mutual TLS
- Locking down the network using network policies
- Supporting external clients with an ingress controller

In the chapters so far, you have learned how to secure user-facing APIs from a variety of threats using security controls such as authentication, authorization, and rate-limiting. It's increasingly common for applications to themselves be structured as a set of microservices, communicating with each other using internal APIs intended to be used by other microservices rather than directly by users. The example in figure 10.1 shows a set of microservices implementing a fictional web store. A single user-facing API provides an interface for a web application, and in turn, calls several backend microservices to handle stock checks, process payment card details, and arrange for products to be shipped once an order is placed.

**DEFINITION** A microservice is an independently deployed service that is a component of a larger application. Microservices are often contrasted with monoliths, where all the components of an application are bundled into a single deployed unit. Microservices communicate with each other using APIs over a protocol such as HTTP.

Some microservices may also need to call APIs provided by external services, such as a third-party payment processor. In this chapter, you'll learn how to securely deploy microservice APIs as Docker containers on Kubernetes, including how to harden containers and the cluster network to reduce the risk of compromise, and how to run TLS at scale using Linkerd (https://linkerd.io) to secure microservice API communications.

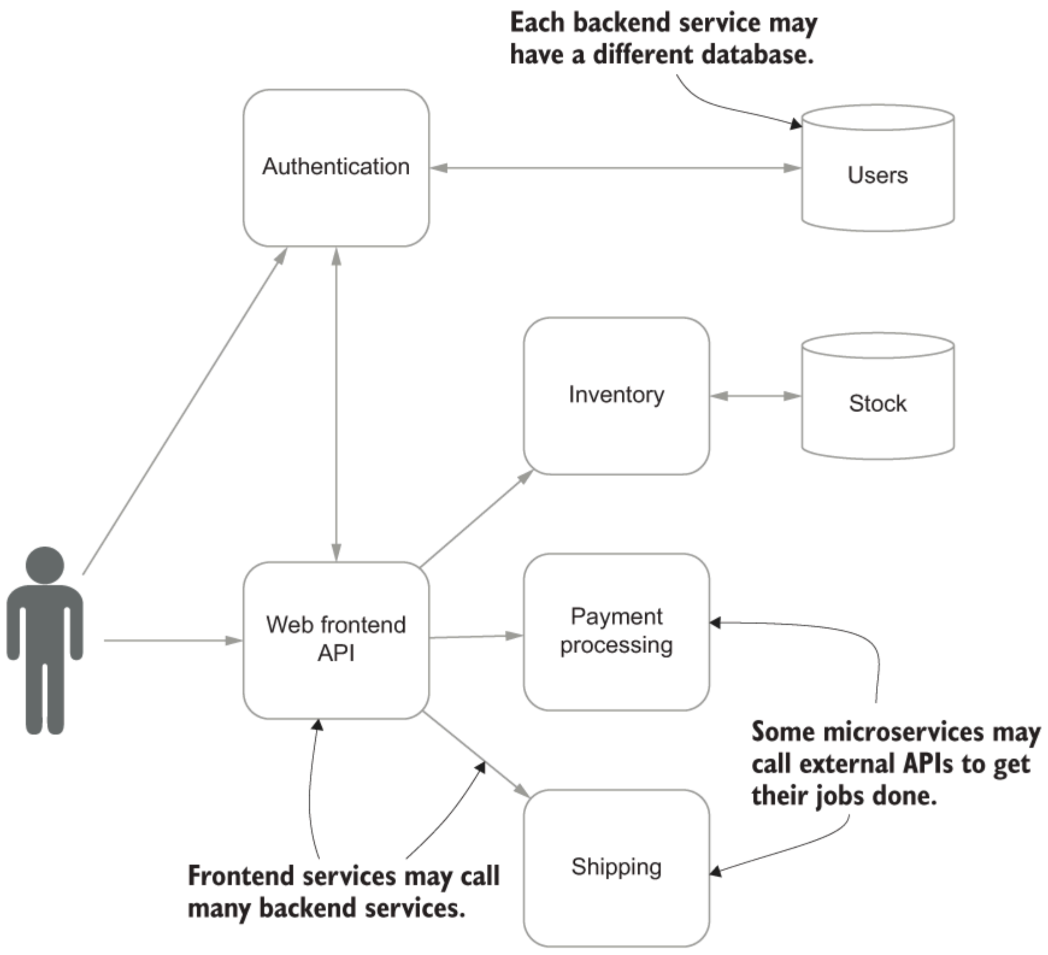
Each backend service may have a different database.
Authentication
Users
Inventory
Stock
Web frontend API
Payment processing
Some microservices may call external APIs to get their jobs done.
Shipping
Frontend services may call many backend services.

Figure 10.1 In a microservices architecture, a single application is broken into loosely coupled services that communicate using remote APIs. In this example, a fictional web store has an API for web clients that calls to internal services to check stock levels, process payments, and arrange shipping when an order is placed.

## 10.1 Microservice APIs on Kubernetes

Although the concepts in this chapter are applicable to most microservice deployments, in recent years the Kubernetes project (https://kubernetes.io) has emerged as a leading approach to deploying and managing microservices in production. To keep things concrete, you'll use Kubernetes to deploy the examples in this part of the book. Appendix B has detailed instructions on how to set up the Minikube environment for running Kubernetes on your development machine. You should follow those instructions now before continuing with the chapter.

The basic concepts of Kubernetes relevant to deploying an API are shown in figure 10.2. A Kubernetes cluster consists of a set of nodes, which are either physical or virtual machines (VMs) running the Kubernetes software. When you deploy an app to the cluster, Kubernetes replicates the app across nodes to achieve availability and scalability requirements that you specify. For example, you might specify that you always require at least three copies of your app to be running, so that if one fails the other two can handle the load. Kubernetes ensures these availability goals are always satisfied and redistributing apps as nodes are added or removed from the cluster. An app is implemented by one or more pods, which encapsulate the software needed to run that app. A pod is itself made up of one or more Linux containers, each typically running a single process such as an HTTP API server.

**DEFINITION** A Kubernetes node is a physical or virtual machine that forms part of the Kubernetes cluster. Each node runs one or more pods that implement apps running on the cluster. A pod is itself a collection of Linux containers and each container runs a single process such as an HTTP server.

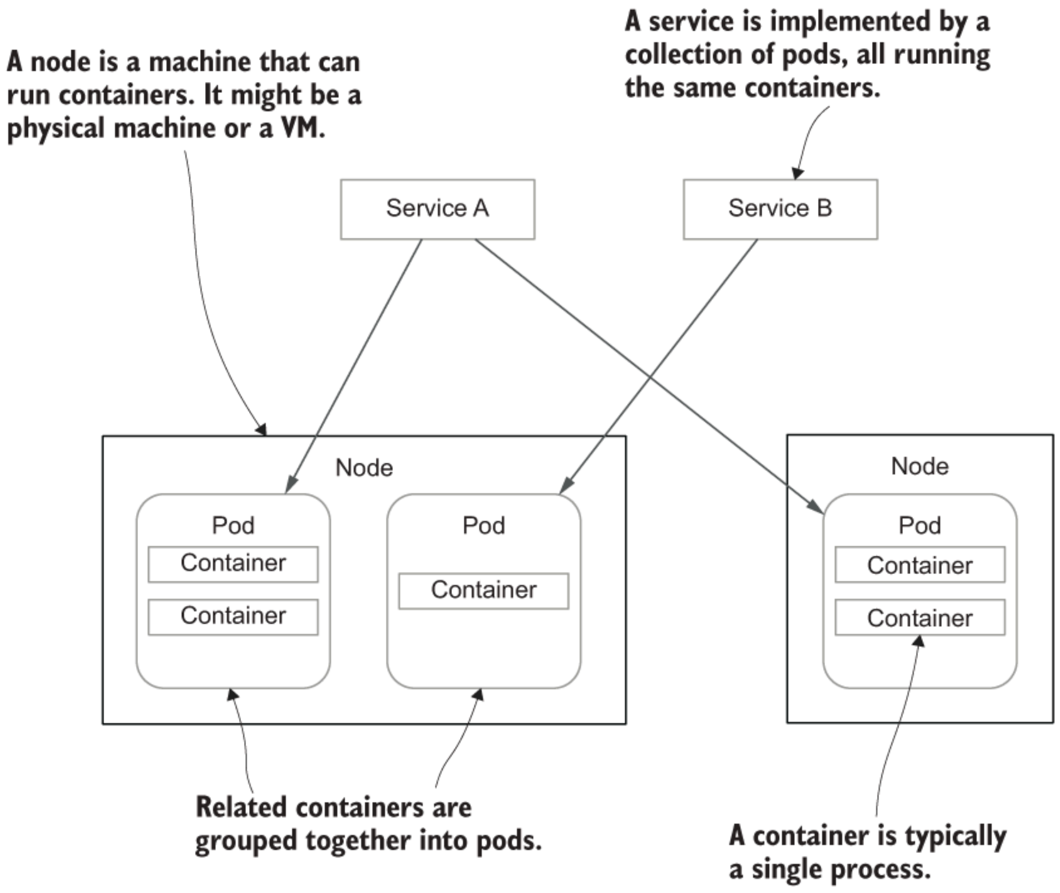
A node is a machine that can run containers. It might be a physical machine or a VM.
A service is implemented by a collection of pods, all running the same containers.
Service A
Service B
Node
Pod
Container
Container
Pod
Container
Node
Pod
Container
Container
Related containers are grouped together into pods.
A container is typically a single process.

Figure 10.2 In Kubernetes, an app is implemented by one or more identical pods running on physical or virtual machines known as nodes. A pod itself is a collection of Linux containers, each of which typically has a single process running within it, such as an API server.

A Linux container is the name given to a collection of technologies within the Linux operating system that allow a process (or collection of processes) to be isolated from other processes so that it sees its own view of the file system, network, users, and other shared resources. This simplifies packaging and deployment, because different processes can use different versions of the same components, which might otherwise cause conflicts. You can even run entirely different distributions of Linux within containers simultaneously on the same operating system kernel. Containers also provide security benefits, because processes can be locked down within a container such that it is much harder for an attacker that compromises one process to break out of the container and affect other processes running in different containers or the host operating system. In this way, containers provide some of the benefits of VMs, but with lower overhead. Several tools for packaging Linux containers have been developed, the most famous of which is Docker (https://www.docker.com), which many Kubernetes deployments build on top of.

**LEARN ABOUT IT** Securing Linux containers is a complex topic, and we'll cover only the basics of in this book. The NCC Group have published a freely available 123-page guide to hardening containers at http://mng.bz/wpQQ.

In most cases, a pod should contain only a single main container and that container should run only a single process. If the process (or node) dies, Kubernetes will restart the pod automatically, possibly on a different node. There are two general exceptions to the one-container-per-pod rule:

- An init container runs before any other containers in the pod and can be used to perform initialization tasks, such as waiting for other services to become available. The main container in a pod will not be started until all init containers have completed.
- A sidecar container runs alongside the main container and provides additional services. For example, a sidecar container might implement a reverse proxy for an API server running in the main container, or it might periodically update data files on a filesystem shared with the main container.

For the most part, you don't need to worry about these different kinds of containers in this chapter and can stick to the one-container-per-pod rule. You'll see an example of a sidecar container when you learn about the Linkerd service mesh in section 10.3.2.

A Kubernetes cluster can be highly dynamic with pods being created and destroyed or moved from one node to another to achieve performance and availability goals. This makes it challenging for a container running in one pod to call an API running in another pod, because the IP address may change depending on what node (or nodes) it happens to be running on. To solve this problem, Kubernetes has the concept of a service, which provides a way for pods to find other pods within the cluster. Each service running within Kubernetes is given a unique virtual IP address that is unique to that service, and Kubernetes keeps track of which pods implement that service. In a microservice architecture, you would register each microservice as a separate Kubernetes service. A process running in a container can call another microservice's API by making a network request to the virtual IP address corresponding to that service. Kubernetes will intercept the request and redirect it to a pod that implements the service.

**DEFINITION** A Kubernetes service provides a fixed virtual IP address that can be used to send API requests to microservices within the cluster. Kubernetes will route the request to a pod that implements the service.

As pods and nodes are created and deleted, Kubernetes updates the service metadata to ensure that requests are always sent to an available pod for that service. A DNS service is also typically running within a Kubernetes cluster to convert symbolic names for services, such as `payments.myapp.svc.example.com`, into its virtual IP address, such as `192.168.0.12`. This allows your microservices to make HTTP requests to hard-coded URIs and rely on Kubernetes to route the request to an appropriate pod. By default, services are accessible internally only within the Kubernetes network, but you can also publish a service to a public IP address either directly or using a reverse proxy or load balancer. You'll learn how to deploy a reverse proxy in section 10.4.

Pop quiz

1. A Kubernetes pod contains which one of the following components?
    1. Node
    2. Service
    3. Container
    4. Service mesh
    5. Namespace
2. True or False: A sidecar container runs to completion before the main container starts.

The answers are at the end of the chapter.

## 10.2 Deploying Natter on Kubernetes

In this section, you’ll learn how to deploy a real API into Kubernetes and how to configure pods and services to allow microservices to talk to each other. You’ll also add a new link-preview microservice as an example of securing microservice APIs that are not directly accessible to external users. After describing the new microservice, you’ll use the following steps to deploy the Natter API to Kubernetes:

1. Building the H2 database as a Docker container.
2. Deploying the database to Kubernetes.
3. Building the Natter API as a Docker container and deploying it.
4. Building the new link-preview microservice.
5. Deploying the new microservice and exposing it as a Kubernetes service.
6. Adjusting the Natter API to call the new microservice API.

You’ll then learn how to avoid common security vulnerabilities that the link-preview microservice introduces and harden the network against common attacks. But first let’s motivate the new link-preview microservice.

You’ve noticed that many Natter users are using the app to share links with each other. To improve the user experience, you’ve decided to implement a feature to generate previews for these links. You’ve designed a new microservice that will extract links from messages and fetch them from the Natter servers to generate a small preview based on the metadata in the HTML returned from the link, making use of any Open Graph tags in the page (**https://ogp.me**). For now, this service will just look for a title, description, and optional image in the page metadata, but in future you plan to expand the service to handle fetching images and videos. You’ve decided to deploy the new link-preview API as a separate microservice, so that an independent team can develop it.

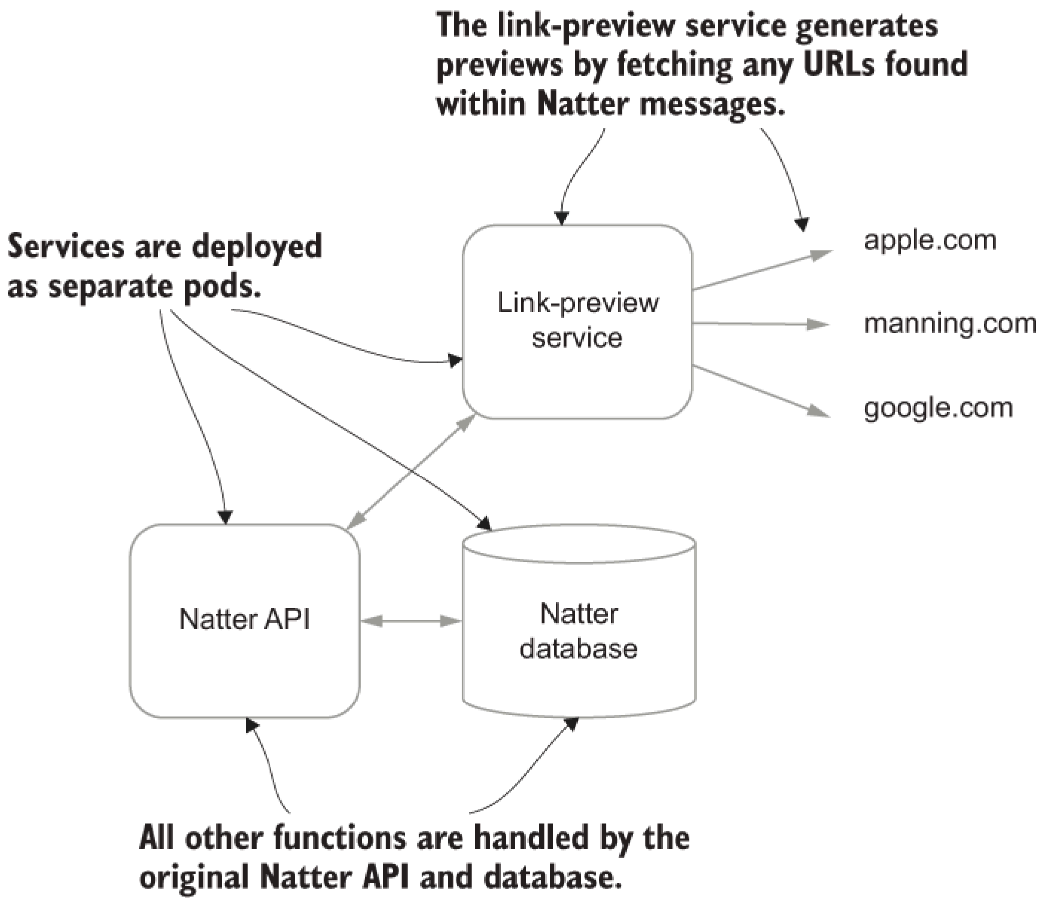
The link-preview service generates previews by fetching any URLs found within Natter messages.
Services are deployed as separate pods.
Link-preview service
apple.com
manning.com
google.com
Natter API
Natter database
All other functions are handled by the original Natter API and database.

Figure 10.3 The link-preview API is developed and deployed as a new microservice, separate from the main Natter API and running in different pods.

Figure 10.3 shows the new deployment, with the existing Natter API and database joined by the new link-preview microservice. Each of the three components is implemented by a separate group of pods, which are then exposed internally as three Kubernetes services:

- The H2 database runs in one pod and is exposed as the natter-database-service.
- The link-preview microservice runs in another pod and provides the natter-link-preview-service.
- The main Natter API runs in yet another pod and is exposed as the natter-api-service.

You’ll use a single pod for each service in this chapter, for simplicity, but Kubernetes allows you to run multiple copies of a pod on multiple nodes for performance and reliability: if a pod (or node) crashes, it can then redirect requests to another pod implementing the same service.

Separating the link-preview service from the main Natter API also has security benefits, because fetching and parsing arbitrary content from the internet is potentially risky. If this was done within the main Natter API process, then any mishandling of those requests could compromise user data or messages. Later in the chapter you'll see examples of attacks that can occur against this link-preview API and how to lock down the environment to prevent them causing any damage. Separating potentially risky operations into their own environments is known as privilege separation.

**DEFINITION** Privilege separation is a design technique based on extracting potentially risky operations into a separate process or environment that is isolated from the main process. The extracted process can be run with fewer privileges, reducing the damage if it is ever compromised.

Before you develop the new link-preview service, you'll get the main Natter API running on Kubernetes with the H2 database running as a separate service.

### 10.2.1 Building H2 database as a Docker container

Although the H2 database you've used for the Natter API in previous chapters is intended primarily for embedded use, it does come with a simple server that can be used for remote access. The first step of running the Natter API on Kubernetes is to build a Linux container for running the database. There are several varieties of Linux container; in this chapter, you'll build a Docker container (as that is the default used by the Minikube environment) to run Kubernetes on a local developer machine. See appendix B for detailed instructions on how to install and configure Docker and Minikube. Docker container images are built using a Dockerfile, which is a script that describes how to build and run the software you need.

**DEFINITION** A container image is a snapshot of a Linux container that can be used to create many identical container instances. Docker images are built in layers from a base image that specifies the Linux distribution such as Ubuntu or Debian. Different containers can share the base image and apply different layers on top, reducing the need to download and store large images multiple times.

Because there is no official H2 database Docker file, you can create your own, as shown in listing 10.1. Navigate to the root folder of the Natter project and create a new folder named docker and then create a folder inside there named h2. Create a new file named Dockerfile in the new docker/h2 folder you just created with the contents of the listing. A Dockerfile consists of the following components:

- A base image, which is typically a Linux distribution such as Debian or Ubuntu. The base image is specified using the `FROM` statement.
- A series of commands telling Docker how to customize that base image for your app. This includes installing software, creating user accounts and permissions, or setting up environment variables. The commands are executed within a container running the base image.

**DEFINITION** A base image is a Docker container image that you use as a starting point for creating your own images. A Dockerfile modifies a base image to install additional dependencies and configure permissions.

The Dockerfile in the listing downloads the latest release of H2, verifies its SHA-256 hash to ensure the file hasn't changed, and unpacks it. The Dockerfile uses `curl` to download the H2 release and `sha256sum` to verify the hash, so you need to use a base image that includes these commands. Docker runs these commands in a container running the base image, so it will fail if these commands are not available, even if you have `curl` and `sha256sum` installed on your development machine.

To reduce the size of the final image and remove potentially vulnerable files, you can then copy the server binaries into a different, minimal base image. This is known as a Docker multistage build and is useful to allow the build process to use a full-featured image while the final image is based on something more stripped-down. This is done in listing 10.1 by adding a second `FROM` command to the Dockerfile, which causes Docker to switch to the new base image. You can then copy files from the build image using the `COPY` `--from` command as shown in the listing.

**DEFINITION** A Docker multistage build allows you to use a full-featured base image to build and configure your software but then switch to a stripped-down base image to reduce the size of the final image.

In this case, you can use Google's distroless base image, which contains just the Java 11 runtime and its dependencies and nothing else (not even a shell). Once you've copied the server files into the base image, you can then expose port 9092 so that the server can be accessed from outside the container and configure it to use a non-root user and group to run the server. Finally, define the command to run to start the server using the `ENTRYPOINT` command.

**TIP** Using a minimal base image such as the Alpine distribution or Google's distroless images reduces the attack surface of potentially vulnerable software and limits further attacks that can be carried out if the container is ever compromised. In this case, an attacker would be quite happy to find curl on a compromised container, but this is missing from the distroless image as is almost anything else they could use to further an attack. Using a minimal image also reduces the frequency with which you'll need to apply security updates to patch known vulnerabilities in the distribution because the vulnerable components are not present.

Listing 10.1 The H2 database Dockerfile

```
FROM curlimages/curl:7.66.0 AS build-env

ENV RELEASE h2-2018-03-18.zip                                              ❶
ENV SHA256 \
    a45e7824b4f54f5d9d65fb89f22e1e75ecadb15ea4dcf8c5d432b80af59ea759      ❶

WORKDIR /tmp

RUN echo "$SHA256  $RELEASE" > $RELEASE.sha256 && \
    curl -sSL https://www.h2database.com/$RELEASE -o $RELEASE && \        ❷
    sha256sum -b -c $RELEASE.sha256 && \                                  ❷
    unzip $RELEASE && rm -f $RELEASE                                      ❸

FROM gcr.io/distroless/java:11                                            ❹
WORKDIR /opt                                                              ❹
COPY --from=build-env /tmp/h2/bin /opt/h2                                 ❹

USER 1000:1000                                                            ❺

EXPOSE 9092                                                               ❻
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/urandom", \            ❼
        "-cp", "/opt/h2/h2-1.4.197.jar", \                                ❼
        "org.h2.tools.Server", "-tcp", "-tcpAllowOthers"]                 ❼
```

❶ Define environment variables for the release file and hash.

❷ Download the release and verify the SHA-256 hash.

❸ Unzip the download and delete the zip file.

❹ Copy the binary files into a minimal container image.

❺ Ensure the process runs as a non-root user and group.

❻ Expose the H2 default TCP port.

❼ Configure the container to run the H2 server.

Linux users and UIDs

When you log in to a Linux operating system (OS) you typically use a string username such as "guest" or "root." Behind the scenes, Linux maps these usernames into 32-bit integer UIDs (user IDs). The same happens with group names, which are mapped to integer GIDs (group IDs). The mapping between usernames and UIDs is done by the /etc/passwd file, which can differ inside a container from the host OS. The root user always has a UID of 0. Normal users usually have UIDs starting at 500 or 1000. All permissions to access files and other resources are determined by the operating system in terms of UIDs and GIDs rather than user and group names, and a process can run with a UID or GID that doesn't correspond to any named user or group.

By default, UIDs and GIDs within a container are identical to those in the host. So UID 0 within the container is the same as UID 0 outside the container: the root user. If you run a process inside a container with a UID that happens to correspond to an existing user in the host OS, then the container process will inherit all the permissions of that user on the host. For added security, your Docker images can create a new user and group and let the kernel assign an unused UID and GID without any existing permissions in the host OS. If an attacker manages to exploit a vulnerability to gain access to the host OS or filesystem, they will have no (or very limited) permissions.

A Linux user namespace can be used to map UIDs within the container to a different range of UIDs on the host. This allows a process running as UID 0 (root) within a container to be mapped to a non-privileged UID such as 20000 in the host. As far as the container is concerned, the process is running as root, but it would not have root privileges if it ever broke out of the container to access the host. See https://docs.docker.com/engine/security/userns-remap/ for how to enable a user namespace in Docker. This is not yet possible in Kubernetes, but there are several alternative options for reducing user privileges inside a pod that are discussed later in the chapter.

When you build a Docker image, it gets cached by the Docker daemon that runs the build process. To use the image elsewhere, such as within a Kubernetes cluster, you must first push the image to a container repository such as Docker Hub (https:// hub.docker.com) or a private repository within your organization. To avoid having to configure a repository and credentials in this chapter, you can instead build directly to the Docker daemon used by Minikube by running the following commands in your terminal shell. You should specify version 1.16.2 of Kubernetes to ensure compatibility with the examples in this book. Some of the examples require Minikube to be running with at least 4GB of RAM, so use the `--memory` flag to specify that.

```
minikube start \
    --kubernetes-version=1.16.2 \     ❶
    --memory=4096                     ❷
```

❶ Enable the latest Kubernetes version.

❷ Specify 4GB of RAM.

You should then run

```
eval $(minikube docker-env)
```

so that any subsequent Docker commands in the same console instance will use Minikube's Docker daemon. This ensures Kubernetes will be able to find the images without needing to access an external repository. If you open a new terminal window, make sure to run this command again to set the environment correctly.

**LEARN ABOUT IT** Typically in a production deployment, you'd configure your DevOps pipeline to automatically push Docker images to a repository after they have been thoroughly tested and scanned for known vulnerabilities. Setting up such a workflow is outside the scope of this book but is covered in detail in Securing DevOps by Julien Vehent (Manning, 2018; http://mng.bz/qN52).

You can now build the H2 Docker image by typing the following commands in the same shell:

```
cd docker/h2
docker build -t apisecurityinaction/h2database .
```

This may take a long time to run the first time because it must download the base images, which are quite large. Subsequent builds will be faster because the images are cached locally. To test the image, you can run the following command and check that you see the expected output:

```
$ docker run apisecurityinaction/h2database
TCP server running at tcp://172.17.0.5:9092 (others can connect)
If you want to stop the container press Ctrl-C.
```

**TIP** If you want to try connecting to the database server, be aware that the IP address displayed is for Minikube's internal virtual networking and is usually not directly accessible. Run the command `minikube ip` at the prompt to get an IP address you can use to connect from the host OS.

### 10.2.2 Deploying the database to Kubernetes

To deploy the database to the Kubernetes cluster, you'll need to create some configuration files describing how it is to be deployed. But before you do that, an important first step is to create a separate Kubernetes namespace to hold all pods and services related to the Natter API. A namespace provides a level of isolation when unrelated services need to run on the same cluster and makes it easier to apply other security policies such as the networking policies that you'll apply in section 10.3. Kubernetes provides several ways to configure objects in the cluster, including namespaces, but it's a good idea to use declarative configuration files so that you can check these into Git or another version-control system, making it easier to review and manage security configuration over time. Listing 10.2 shows the configuration needed to create a new namespace for the Natter API. Navigate to the root folder of the Natter API project and create a new sub-folder named "kubernetes." Then inside the folder, create a new file named natter-namespace.yaml with the contents of listing 10.2. The file tells Kubernetes to make sure that a namespace exists with the name `natter-api` and a matching label.

**WARNING** YAML (https://yaml.org) configuration files are sensitive to indentation and other whitespace. Make sure you copy the file exactly as it is in the listing. You may prefer to download the finished files from the GitHub repository accompanying the book (http://mng.bz/7Gly).

Listing 10.2 Creating the namespace

```
apiVersion: v1
kind: Namespace            ❶
metadata:
    name: natter-api        ❷
    labels:                 ❷
      name: natter-api      ❷
```

❶ Use the Namespace kind to create a namespace.

❷ Specify a name and label for the namespace.

**NOTE** Kubernetes configuration files are versioned using the `apiVersion` attribute. The exact version string depends on the type of resource and version of the Kubernetes software you're using. Check the Kubernetes documentation (**https://kubernetes.io/docs/home/**) for the correct `apiVersion` when writing a new configuration file.

To create the namespace, run the following command in your terminal in the root folder of the natter-api project:

```
kubectl apply -f kubernetes/natter-namespace.yaml
```

The `kubectl` `apply` command instructs Kubernetes to make changes to the cluster to match the desired state specified in the configuration file. You'll use the same command to create all the Kubernetes objects in this chapter. To check that the namespace is created, use the `kubectl` `get` `namespaces` command:

```
$ kubectl get namespaces
```

Your output will look similar to the following:

```
NAME              STATUS   AGE
default           Active   2d6h
kube-node-lease   Active   2d6h
kube-public       Active   2d6h
kube-system       Active   2d6h
natter-api        Active   6s
```

You can now create the pod to run the H2 database container you built in the last section. Rather than creating the pod directly, you'll instead create a deployment, which describes which pods to run, how many copies of the pod to run, and the security attributes to apply to those pods. Listing 10.3 shows a deployment configuration for the H2 database with a basic set of security annotations to restrict the permissions of the pod in case it ever gets compromised. First you define the name and namespace to run the deployment in, making sure to use the namespace that you defined earlier. A deployment specifies the pods to run by using a selector that defines a set of labels that matching pods will have. In listing 10.3, you define the pod in the template section of the same file, so make sure the labels are the same in both parts.

**NOTE** Because you are using an image that you built directly to the Minikube Docker daemon, you need to specify `imagePullPolicy: Never` in the container specification to prevent Kubernetes trying to pull the image from a repository. In a real deployment, you would have a repository, so you'd remove this setting.

You can also specify a set of standard security attributes in the `securityContext` section for both the pod and for individual containers, as shown in the listing. In this case, the definition ensures that all containers in the pod run as a non-root user, and that it is not possible to bypass the default permissions by setting the following properties:

- `runAsNonRoot : true` ensures that the container is not accidentally run as the root user. The root user inside a container is the root user on the host OS and can sometimes escape from the container.
- `allowPrivilegeEscalation : false` ensures that no process run inside the container can have more privileges than the initial user. This prevents the container executing files marked with set-UID attributes that run as a different user, such as root.
- `readOnlyRootFileSystem : true` makes the entire filesystem inside the container read-only, preventing an attacker from altering any system files. If your container needs to write files, you can mount a separate persistent storage volume.
- `capabilities : drop: - all` removes all Linux capabilities assigned to the container. This ensures that if an attacker does gain root access, they are severely limited in what they can do. Linux capabilities are subsets of full root privileges and are unrelated to the capabilities you used in chapter 9.

**LEARN ABOUT IT** For more information on configuring the security context of a pod, refer to http://mng.bz/mN12. In addition to the basic attributes specified here, you can enable more advanced sandboxing features such as AppArmor, SELinux, or seccomp. These features are beyond the scope of this book. A starting point to learn more is the Kubernetes

Security Best Practices talk given by Ian Lewis at Container Camp 2018 (https://www.youtube.com/watch?v=v6a37uzFrCw).

Create a file named natter-database-deployment.yaml in the kubernetes folder with the contents of listing 10.3 and save the file.

**Listing 10.3 The database deployment**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: natter-database-deployment                        ❶
  namespace: natter-api                                   ❶
spec:
  selector:                                               ❷
    matchLabels:                                          ❷
      app: natter-database                                ❷
  replicas: 1                                             ❸
  template:
    metadata:
      labels:                                             ❷
        app: natter-database                              ❷
    spec:
      securityContext:                                    ❹
        runAsNonRoot: true                                ❹
      containers:
        - name: natter-database                           ❺
          image: apisecurityinaction/h2database:latest    ❺
          imagePullPolicy: Never                          ❻
          securityContext:                                ❹
            allowPrivilegeEscalation: false               ❹
            readOnlyRootFilesystem: true                  ❹
            capabilities:                                 ❹
              drop:                                       ❹
                - all                                     ❹
          ports:                                          ❼
            - containerPort: 9092                         ❼
```

❶ Give the deployment a name and ensure it runs in the natter-api namespace.

❷ Select which pods are in the deployment.

❸ Specify how many copies of the pod to run on the cluster.

❹ Specify a security context to limit permissions inside the containers.

❺ Tell Kubernetes the name of the Docker image to run.

❻ Ensure that Kubernetes uses the local image rather than trying to pull one from a repository.

❼ Expose the database server port to other pods.

Run `kubectl apply -f kubernetes/natter-database-deployment.yaml` in the natter-api root folder to deploy the application.

To check that your pod is now running, you can run the following command:

```
$ kubectl get deployments --namespace=natter-api
```

This will result in output like the following:

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
natter-database-deployment   1/1     1            1           10s
```

You can then check on individual pods in the deployment by running the following command

```
$ kubectl get pods --namespace=natter-api
```

which outputs a status report like this one, although the pod name will be different because Kubernetes generates these randomly:

```
NAME                                          READY   STATUS    RESTARTS   A
natter-database-deployment-8649d65665-d58wb   1/1     Running   0          1
```

Although the database is now running in a pod, pods are designed to be ephemeral and can come and go over the lifetime of the cluster. To provide a stable reference for other pods to connect to, you need to also define a Kubernetes service. A service provides a stable internal IP address and DNS name that other pods can use to connect to the service. Kubernetes will route these requests to an available pod that implements the service. Listing 10.4 shows the service definition for the database.

First you need to give the service a name and ensure that it runs in the natter-api namespace. You define which pods are used to implement the service by defining a selector that matches the label of the pods defined in the deployment. In this case, you used the label `app: natter-database` when you defined the deployment, so use the same label here to make sure the pods are found. Finally, you tell Kubernetes which ports to expose for the service. In this case, you can expose port 9092. When a pod tries to connect to the service on port 9092, Kubernetes will forward the request to the same port on one of the pods that implements the service. If you want to use a different port, you can use the `targetPort` attribute to create a mapping between the service port and the port exposed by the pods. Create a new file named natter-database-service.yaml in the kubernetes folder with the contents of listing 10.4.

Listing 10.4 The database service

```
apiVersion: v1
kind: Service
metadata:
  name: natter-database-service          ❶
  namespace: natter-api                  ❶
spec:
  selector:                              ❷
    app: natter-database                 ❷
  ports:
    - protocol: TCP                      ❸
      port: 9092                         ❸
```

❶ Give the service a name in the natter-api namespace.

❷ Select the pods that implement the service using labels.

❸ Expose the database port.

Run

```
kubectl apply -f kubernetes/natter-database-service.yaml
```

to configure the service.

Pop quiz

3. Which of the following are best practices for securing containers in Kubernetes? Select all answers that apply.
   1. Running as a non-root user
   2. Disallowing privilege escalation
   3. Dropping all unused Linux capabilities
   4. Marking the root filesystem as read-only
   5. Using base images with the most downloads on Docker Hub
   6. Applying sandboxing features such as AppArmor or seccomp

The answer is at the end of the chapter.

### 10.2.3 Building the Natter API as a Docker container

For building the Natter API container, you can avoid writing a Dockerfile manually and make use of one of the many Maven plugins that will do this for you automatically. In this chapter, you'll use the Jib plugin from Google (**https://github.com/GoogleContainerTools/jib**), which requires a minimal amount of configuration to build a container image.

Listing 10.5 shows how to configure the maven-jib-plugin to build a Docker container image for the Natter API. Open the pom.xml file in your editor and add the whole `build` section from listing 10.5 to the bottom of the file just before the closing `</project>` tag. The configuration instructs Maven to include the Jib plugin in the build process and sets several configuration options:

- Set the name of the output Docker image to build to "apisecurityinaction/ natter-api."
- Set the name of the base image to use. In this case, you can use the distroless Java 11 image provided by Google, just as you did for the H2 Docker image.
- Set the name of the main class to run when the container is launched. If there is only one main method in your project, then you can leave this out.
- Configure any additional JVM settings to use when starting the process. The default settings are fine, but as discussed in chapter 5, it is worth telling Java to prefer to use the `/dev/urandom` device for seeding `SecureRandom` instances to avoid potential performance issues. You can do this by setting the `java.security` `.egd` system property.
- Configure the container to expose port 4567, which is the default port that our API server will listen to for HTTP connections.

- Finally, configure the container to run processes as a non-root user and group. In this case you can use a user with UID (user ID) and GID (group ID) of 1000.

Listing 10.5 Enabling the Jib Maven plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>com.google.cloud.tools</groupId>
      <artifactId>jib-maven-plugin</artifactId>
      <version>2.4.0</version>
      <configuration>
        <to>
          <image>apisecurityinaction/natter-api</image>
        </to>
        <from>
          <image>gcr.io/distroless/java:11</image>
        </from>
        <container>
          <mainClass>${exec.mainClass}</mainClass>
          <jvmFlags>
            <jvmFlag>-Djava.security.egd=file:/dev/urandom</jvmFlag>
          </jvmFlags>
          <ports>
            <port>4567</port>
          </ports>
          <user>1000:1000</user>
        </container>
      </configuration>
    </plugin>
  </plugins>
</build>
```

❶ Use the latest version of the jib-maven-plugin.

❷ Provide a name for the generated Docker image.

❸ Use a minimal base image to reduce the size and attack surface.

❹ Specify the main class to run.

❺ Add any custom JVM settings.

❻ Expose the port that the API server listens to so that clients can connect.

❼ Specify a non-root user and group to run the process.

Before you build the Docker image, you should first disable TLS because this avoids configuration issues that will need to be resolved to get TLS working in the cluster. You will learn how to re-enable TLS between microservices in section 10.3. Open Main.java in your editor and find the call to the `secure()` method. Comment out (or delete) the method call as follows:

```
//secure("localhost.p12", "changeit", null, null);     ❶
```

❶ Comment out the secure() method to disable TLS.

The API will still need access to the keystore for any HMAC or AES encryption keys. To ensure that the keystore is copied into the Docker image, navigate to the src/main folder in the project and create a new folder named “jib.” Copy the keystore.p12 file from the root of the project to the src/main/jib folder you just created. The jib-maven-plugin will automatically copy files in this folder into the Docker image it creates.

**WARNING** Copying the keystore and keys directly into the Docker image is poor security because anyone who downloads the image can access your secret keys. In chapter 11, you’ll see how to avoid including the keystore in this way and ensure that you use unique keys for each environment that your API runs in.

You also need to change the JDBC URL that the API uses to connect to the database. Rather than creating a local in-memory database, you can instruct the API to connect to the H2 database service you just deployed. To avoid having to create a disk volume to store data files, in this example you’ll continue using an in-memory database running on the database pod. This is as simple as replacing the current JDBC database URL with the following one, using the DNS name of the database service you created earlier:

```
jdbc:h2:tcp://natter-database-service:9092/mem:natter
```

Open the Main.java file and replace the existing JDBC URL with the new one in the code that creates the database connection pool. The new code should look as shown in listing 10.6.

Listing 10.6 Connecting to the remote H2 database

```
var jdbcUrl =                                                      ❶
    "jdbc:h2:tcp://natter-database-service:9092/mem:natter";      ❶
var datasource = JdbcConnectionPool.create(
    jdbcUrl, "natter", "password");                               ❷
createTables(datasource.getConnection());
datasource = JdbcConnectionPool.create(
    jdbcUrl, "natter_api_user", "password");                      ❷
var database = Database.forDataSource(datasource);
```

❶ Use the DNS name of the remote database service.

❷ Use the same JDBC URL when creating the schema and when switching to the Natter API user.

To build the Docker image for the Natter API with Jib, you can then simply run the following Maven command in the same shell in the root folder of the natter-api project:

```
mvn clean compile jib:dockerBuild
```

You can now create a deployment to run the API in the cluster. Listing 10.7 shows the deployment configuration, which is almost identical to the H2 database deployment you created in the last section. Apart from specifying a different Docker image to run, you should also make sure you attach a different label to the pods that form this deployment. Otherwise, the new pods will be included in the database deployment. Create a new file named natter-api-deployment.yaml in the kubernetes folder with the contents of the listing.

Listing 10.7 The Natter API deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: natter-api-deployment                    ❶
  namespace: natter-api
```

```
spec:
  selector:
    matchLabels:
      app: natter-api                                              ❷
  replicas: 1
  template:
    metadata:
      labels:
        app: natter-api                                            ❷
    spec:
      securityContext:
        runAsNonRoot: true
      containers:
        - name: natter-api
          image: apisecurityinaction/natter-api:latest             ❸
          imagePullPolicy: Never
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop:
                - all
          ports:
            - containerPort: 4567                                  ❹
```

❶ Give the API deployment a unique name.

❷ Ensure the labels for the pods are different from the database pod labels.

❸ Use the Docker image that you built with Jib.

❹ Expose the port that the server runs on.

Run the following command to deploy the code:

```
kubectl apply -f kubernetes/natter-api-deployment.yaml
```

The API server will start and connect to the database service.

The last step is to also expose the API as a service within Kubernetes so that you can connect to it. For the database service, you didn't specify a service type so Kubernetes deployed it using the default ClusterIP type. Such services are only accessible within the cluster, but you want the API to be accessible from external clients, so you need to pick a different service type. The simplest alternative is the NodePort service type, which exposes the service on a port on each node in the cluster. You can then connect to the service using the external IP address of any node in the cluster.

Use the `nodePort` attribute to specify which port the service is exposed on, or leave it blank to let the cluster pick a free port. The exposed port must be in the range 30000-32767. In section 10.4, you'll deploy an ingress controller for a more controlled approach to allowing connections from external clients. Create a new file named natter-api-service.yaml in the kubernetes folder with the contents of listing 10.8.

Listing 10.8 Exposing the API as a service

```
apiVersion: v1
kind: Service
metadata:
  name: natter-api-service
  namespace: natter-api
spec:
  type: NodePort              ❶
  selector:
    app: natter-api
  ports:
    - protocol: TCP
      port: 4567
      nodePort: 30567         ❷
```

❶ Specify the type as NodePort to allow external connections.

❷ Specify the port to expose on each node; it must be in the range 30000-32767.

Now run the command `kubectl apply -f kubernetes/natter-api-service.yaml` to start the service. You can then run the following to get a URL that you can use with curl to interact with the service:

```
$ minikube service --url natter-api-service --namespace=natter-api
```

This will produce output like the following:

```
http://192.168.99.109:30567
```

You can then use that URL to access the API as in the following example:

```
$ curl -X POST -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  http://192.168.99.109:30567/users
{"username":"test"}
```

You now have the API running in Kubernetes.

### 10.2.4 The link-preview microservice

You have Docker images for the Natter API and the H2 database deployed and running in Kubernetes, so it's now time to develop the link-preview microservice. To simplify development, you can create the new microservice within the existing Maven project and reuse the existing classes.

**NOTE** The implementation in this chapter is extremely naïve from a performance and scalability perspective and is intended only to demonstrate API security techniques within Kubernetes.

To implement the service, you can use the jsoup library (https://jsoup.org) for Java, which simplifies fetching and parsing HTML pages. To include jsoup in the project, open the pom.xml file in your editor and add the following lines to the `<dependencies>` section:

```
    <dependency>
      <groupId>org.jsoup</groupId>
      <artifactId>jsoup</artifactId>
      <version>1.13.1</version>
    </dependency>
```

An implementation of the microservice is shown in listing 10.9. The API exposes a single operation, implemented as a GET request to the `/preview` endpoint with the URL from the link as a query parameter. You can use jsoup to fetch the URL and parse the HTML that is returned. Jsoup does a good job of ensuring the URL is a valid HTTP or HTTPS URL, so you can skip performing those checks yourself and instead register Spark exception handlers to return an appropriate response if the URL is invalid or cannot be fetched for any reason.

**WARNING** If you process URLs in this way, you should ensure that an attacker can't submit `file://` URLs and use this to access protected files on the API server disk. Jsoup strictly validates that the URL scheme is HTTP before loading any resources, but if you use a different library you should check the documentation or perform your own validation.

After `jsoup` fetches the HTML page, you can use the `selectFirst` method to find metadata tags in the document. In this case, you're interested in the following tags:

- The document title.
- The Open Graph `description` property, if it exists. This is represented in the HTML as a `<meta>` tag with the `property` attribute set to `og:description`.
- The Open Graph `image` property, which will provide a link to a thumbnail image to accompany the preview.

You can also use the `doc.location()` method to find the URL that the document was finally fetched from just in case any redirects occurred. Navigate to the src/main/ java/com/manning/apisecurityinaction folder and create a new file named LinkPreviewer.java. Copy the contents of listing 10.9 into the file and save it.

**WARNING** This implementation is vulnerable to server-side request forgery (SSRF) attacks. You'll mitigate these issues in section 10.2.7.

Listing 10.9 The link-preview microservice

```
package com.manning.apisecurityinaction;

import java.net.*;

import org.json.JSONObject;
import org.jsoup.Jsoup;
import org.slf4j.*;
import spark.ExceptionHandler;
import static spark.Spark.*;
public class LinkPreviewer {
    private static final Logger logger =
            LoggerFactory.getLogger(LinkPreviewer.class);

    public static void main(String...args) {
        afterAfter((request, response) -> {
            response.type("application/json; charset=utf-8");
        });
        get("/preview", (request, response) -> {
            var url = request.queryParams("url");
            var doc = Jsoup.connect(url).timeout(3000).get();
            var title = doc.title();
            var desc = doc.head()
                    .selectFirst("meta[property='og:description']");
            var img = doc.head()
                    .selectFirst("meta[property='og:image']");

            return new JSONObject()
                    .put("url", doc.location())
                    .putOpt("title", title)
                    .putOpt("description",
                        desc == null ? null : desc.attr("content"))
                    .putOpt("image",
                        img == null ? null : img.attr("content"));
        });

        exception(IllegalArgumentException.class, handleException(400));
        exception(MalformedURLException.class, handleException(400));
        exception(Exception.class, handleException(502));
        exception(UnknownHostException.class, handleException(404));
    }

    private static <T extends Exception> ExceptionHandler<T>
            handleException(int status) {
        return (ex, request, response) -> {
            logger.error("Caught error {} - returning status {}",
                ex, status);
            response.status(status);
            response.body(new JSONObject()
                .put("status", status).toString());
        };
    }
}
```

❶ Because this service will only be called by other services, you can omit the browser security headers.

❷ Extract metadata properties from the HTML.

❸ Produce a JSON response, taking care with attributes that might be null.

❹ Return appropriate HTTP status codes if jsoup raises an exception.

### 10.2.5 Deploying the new microservice

To deploy the new microservice to Kubernetes, you need to first build the link-preview microservice as a Docker image, and then create a new Kubernetes deployment and service configuration for it. You can reuse the existing jib-maven-plugin the build the Docker image, overriding the image name and main class on the command line. Open a terminal in the root folder of the Natter API project and run the following commands to build the image to the Minikube Docker daemon. First, ensure the environment is configured correctly by running:

```
eval $(minikube docker-env)
```

Then use Jib to build the image for the link-preview service:

```
mvn clean compile jib:dockerBuild \
  -Djib.to.image=apisecurityinaction/link-preview \
  -Djib.container.mainClass=com.manning.apisecurityinaction.
➥ LinkPreviewer
```

You can then deploy the service to Kubernetes by applying a deployment configuration, as shown in listing 10.10. This is a copy of the deployment configuration used for the main Natter API, with the pod names changed and updated to use the Docker image that you just built. Create a new file named kubernetes/natter-link-preview-deployment.yaml using the contents of listing 10.10.

Listing 10.10 The link-preview service deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: link-preview-service-deployment
  namespace: natter-api
spec:
  selector:
    matchLabels:
      app: link-preview-service                    [1]
  replicas: 1
  template:
    metadata:
      labels:
        app: link-preview-service                  [1]
    spec:
      securityContext:
        runAsNonRoot: true
      containers:
        - name: link-preview-service
          image: apisecurityinaction/link-preview-service:latest    2
          imagePullPolicy: Never
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop:
                - all
          ports:
            - containerPort: 4567
```

1 Give the pods the name link-preview-service.

2 Use the link-preview-service Docker image you just built.

Run the following command to create the new deployment:

```
kubectl apply -f \
  kubernetes/natter-link-preview-deployment.yaml
```

To allow the Natter API to locate the new service, you should also create a new Kubernetes service configuration for it. Listing 10.11 shows the configuration for the new service, selecting the pods you just created and exposing port 4567 to allow access to the API. Create the file kubernetes/natter-link-preview-service.yaml with the contents of the new listing.

Listing 10.11 The link-preview service configuration

```
apiVersion: v1
kind: Service
metadata:
  name: natter-link-preview-service     1
  namespace: natter-api
spec:
  selector:
    app: link-preview                   2
  ports:
    - protocol: TCP                 ❸
      port: 4567                    ❸
```

❶ Give the service a name.

❷ Make sure to use the matching label for the deployment pods.

❸ Expose port 4567 that the API will run on.

Run the following command to expose the service within the cluster:

```
kubectl apply -f kubernetes/natter-link-preview-service.yaml
```

### 10.2.6 Calling the link-preview microservice

The ideal place to call the link-preview service is when a message is initially posted to the Natter API. The preview data can then be stored in the database along with the message and served up to all users. For simplicity, you can instead call the service when reading a message. This is very inefficient because the preview will be regenerated every time the message is read, but it is convenient for the purpose of demonstration.

The code to call the link-preview microservice is shown in listing 10.12. Open the SpaceController.java file and add the following imports to the top:

```
import java.net.*;
import java.net.http.*;
import java.net.http.HttpResponse.BodyHandlers;
import java.nio.charset.StandardCharsets;
import java.util.*;
import java.util.regex.Pattern;
```

Then add the fields and new method defined in the listing. The new method takes a link, extracted from a message, and calls the link-preview service passing the link URL as a query parameter. If the response is successful, then it returns the link-preview JSON.

Listing 10.12 Fetching a link preview

```
  private final HttpClient httpClient = HttpClient.newHttpClient();     ❶
  private final URI linkPreviewService = URI.create(                    ❶
            "http://natter-link-preview-service:4567");          1

    private JSONObject fetchLinkPreview(String link) {
        var url = linkPreviewService.resolve("/preview?url=" +    2
                URLEncoder.encode(link, StandardCharsets.UTF_8));  2
        var request = HttpRequest.newBuilder(url)                 2
                .GET()                                            2
                .build();                                         2
        try {
            var response = httpClient.send(request,
                    BodyHandlers.ofString());
            if (response.statusCode() == 200) {                   3
              return new JSONObject(response.body());             3
            }
        } catch (Exception ignored) { }
        return null;
    }
```

1 Construct a HttpClient and a constant for the microservice URI.

2 Create a GET request to the service, passing the link as the url query parameter.

3 If the response is successful, then return the JSON link preview.

To return the links from the Natter API, you need to update the `Message` class used to represent a message read from the database. In the SpaceController.java file, find the `Message` class definition and update it to add a new `links` field containing a list of link previews, as shown in listing 10.13.

**TIP** If you haven't added support for reading messages to the Natter API, you can download a fully implemented API from the GitHub repository accompanying the book: https://github.com/NeilMadden/apisecurityinaction. Check out the chapter10 branch for a starting point, or chapter10-end for the completed code.

Listing 10.13 Adding links to a message

```
    public static class Message {
      private final long spaceId;
      private final long msgId;
      private final String author;
      private final Instant time;
    private final String message;
    private final List<JSONObject> links = new ArrayList<>();      ❶

    public Message(long spaceId, long msgId, String author,
        Instant time, String message) {
      this.spaceId = spaceId;
      this.msgId = msgId;
      this.author = author;
      this.time = time;
      this.message = message;
    }
    @Override
    public String toString() {
      JSONObject msg = new JSONObject();
      msg.put("uri",
          "/spaces/" + spaceId + "/messages/" + msgId);
      msg.put("author", author);
      msg.put("time", time.toString());
      msg.put("message", message);
      msg.put("links", links);                                   ❷
      return msg.toString();
    }
  }
```

❶ Add a list of link previews to the class.

❷ Return the links as a new field on the message response.

Finally, you can update the `readMessage` method to scan the text of a message for strings that look like URLs and fetch a link preview for those links. You can use a regular expression to search for potential links in the message. In this case, you'll just look for any strings that start with http:// or https://, as shown in listing 10.14. Once a potential link has been found, you can use the `fetchLinkPreview` method you just wrote to fetch the link preview. If the link was valid and a preview was returned, then add the preview to the list of links on the message. Update the `readMessage` method in the SpaceController.java file to match listing 10.14. The new code is highlighted in bold.

Listing 10.14 Scanning messages for links

```
  public Message readMessage(Request request, Response response) {
    var spaceId = Long.parseLong(request.params(":spaceId"));
    var msgId = Long.parseLong(request.params(":msgId"));
```

```
        var message = database.findUnique(Message.class,
            "SELECT space_id, msg_id, author, msg_time, msg_text " +
                "FROM messages WHERE msg_id = ? AND space_id = ?",
            msgId, spaceId);

        var linkPattern = Pattern.compile("https?://\\S+");      1
        var matcher = linkPattern.matcher(message.message);      1
        int start = 0;
        while (matcher.find(start)) {                            3
            var url = matcher.group();                           3
            var preview = fetchLinkPreview(url);                 3
            if (preview != null) {
                message.links.add(preview);                      4
            }
            start = matcher.end();
        }

        response.status(200);
        return message;
      }
```

1 Use a regular expression to find links in the message.

2 Send each link to the link-preview service.

3 If it was valid, then add the link preview to the links list in the message.

You can now rebuild the Docker image by running the following command in a terminal in the root folder of the project (make sure to set up the Docker environment again if this is a new terminal window):

```
mvn clean compile jib:dockerBuild
```

Because the image is not versioned, Minikube won't automatically pick up the new image. The simplest way to use the new image is to restart Minikube, which will reload all the images from the Docker daemon:[1]

```
minikube stop
```

and then

```
minikube start
```

You can now try out the link-preview service. Use the `minikube ip` command to get the IP address to use to connect to the service. First create a user:

```
curl http://$(minikube ip):30567/users \
  -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}'
```

Next, create a social space and extract the message read-write capability URI into a variable:

```
MSGS_URI=$(curl http://$(minikube ip):30567/spaces \
  -H 'Content-Type: application/json' \
  -d '{"owner":"test","name":"test space"}' \
  -u test:password | jq -r '."messages-rw"')
```

You can now create a message with a link to a HTML story in it:

```
MSG_LINK=$(curl http://$(minikube ip):30567$MSGS_URI \
  -u test:password \
  -H 'Content-Type: application/json' \
  -d '{"author":"test", "message":"Check out this link:
➥   http://www.bbc.co.uk/news/uk-scotland-50435811"}' | jq -r .uri)
```

Finally, you can retrieve the message to see the link preview:

```
curl -u test:password http://$(minikube ip):30567$MSG_LINK | jq
```

The output will look like the following:

```
{
  "author": "test",
  "links": [
    {
      "image":
➥  "https://ichef.bbci.co.uk/news/1024/branded_news/128FC/
➥  production/_109682067_brash_tracks_on_fire_dyke_2019.
➥  creditpaulturner.jpg",
        "description": "The massive fire in the Flow Country in May
➥ doubled Scotland's greenhouse gas emissions while it burnt.",
        "title": "Huge Flow Country wildfire 'doubled Scotland's
➥ emissions' - BBC News",
        "url": "https://www.bbc.co.uk/news/uk-scotland-50435811"
      }
    ],
    "time": "2019-11-18T10:11:24.944Z",
    "message": "Check out this link:
➥ http://www.bbc.co.uk/news/uk-scotland-50435811"
  }
```

### 10.2.7 Preventing SSRF attacks

The link-preview service currently has a large security flaw, because it allows anybody to submit a message with a link that will then be loaded from inside the Kubernetes network. This opens the application up to a server-side request forgery (SSRF) attack, where an attacker crafts a link that refers to an internal service that isn't accessible from outside the network, as shown in figure 10.4.

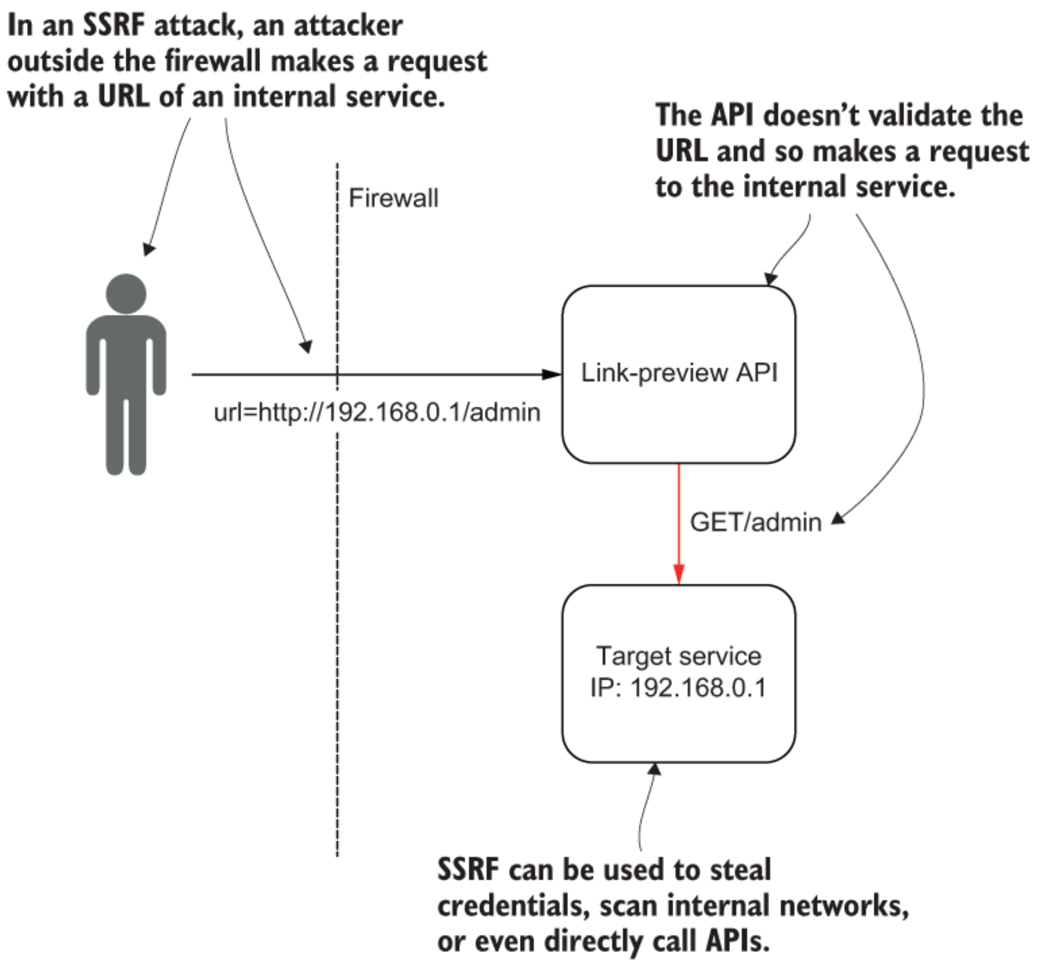
In an SSRF attack, an attacker outside the firewall makes a request with a URL of an internal service.
Firewall
The API doesn't validate the URL and so makes a request to the internal service.
Link-preview API
url=http://192.168.0.1/admin
GET/admin
Target service IP: 192.168.0.1
SSRF can be used to steal credentials, scan internal networks, or even directly call APIs.

Figure 10.4 In an SSRF attack, the attacker sends a URL to a vulnerable API that refers to an internal service. If the API doesn't validate the URL, it will make a request to the internal service that the attacker couldn't

make themselves. This may allow the attacker to probe internal services for vulnerabilities, steal credentials returned from these endpoints, or directly cause actions via vulnerable APIs.

**DEFINITION** A server-side request forgery attack occurs when an attacker can submit URLs to an API that are then loaded from inside a trusted network. By submitting URLs that refer to internal IP addresses the attacker may be able to discover what services are running inside the network or even to cause side effects.

SSRF attacks can be devastating in some cases. For example, in July 2019, Capital One, a large financial services company, announced a data breach that compromised user details, Social Security numbers, and bank account numbers (**http://mng.bz/6AmD**). Analysis of the attack (**https://ejj.io/blog/capital-one**) showed that the attacker exploited a SSRF vulnerability in a Web Application Firewall to extract credentials from the AWS metadata service, which is exposed as a simple HTTP server available on the local network. These credentials were then used to access secure storage buckets containing the user data.

Although the AWS metadata service was attacked in this case, it is far from the first service to assume that requests from within an internal network are safe. This used to be a common assumption for applications installed inside a corporate firewall, and you can still find applications that will respond with sensitive data to completely unauthenticated HTTP requests. Even critical elements of the Kubernetes control plane, such as the etcd database used to store cluster configuration and service credentials, can sometimes be accessed via unauthenticated HTTP requests (although this is usually disabled). The best defense against SSRF attacks is to require authentication for access to any internal services, regardless of whether the request originated from an internal network: an approach known as zero trust networking.

**DEFINITION** A zero trust network architecture is one in which requests to services are not trusted purely because they come from an internal network. Instead, all API requests should be actively authenticated using techniques such as those described in this book. The term originated with Forrester Research and was popularized by Google's BeyondCorp enterprise architecture (**https://cloud.google.com/beyondcorp/**). The term has now become a marketing buzzword, with many products promising a zero-trust approach, but the core idea is still valuable.

Although implementing a zero-trust approach throughout an organization is ideal, this can't always be relied upon, and a service such as the link-preview microservice shouldn't assume that all requests are safe. To prevent the link-preview service being abused for SSRF attacks, you should validate URLs passed to the service before making a HTTP request. This validation can be done in two ways:

- You can check the URLs against a set of allowed hostnames, domain names, or (ideally) strictly match the entire URL. Only URLs that match the allowlist are allowed. This approach is the most secure but is not always feasible.
- You can block URLs that are likely to be internal services that should be protected. This is less secure than allowlisting for several reasons. First, you may forget to blocklist some services. Second, new services may be added later without the blocklist being updated. Blocklisting should only be used when allowlisting is not an option.

For the link-preview microservice, there are too many legitimate websites to have a hope of listing them all, so you'll fall back on a form of blocklisting: extract the hostname from the URL and then check that the IP address does not resolve to a private IP address. There are several classes of IP addresses that are never valid targets for a link-preview service:

- Any loopback address, such as 127.0.0.1, which always refers to the local machine. Allowing requests to these addresses might allow access to other containers running in the same pod.
- Any link-local IP address, which are those starting 169.254 in IPv4 or fe80 in IPv6. These addresses are reserved for communicating with hosts on the same network segment.
- Private-use IP address ranges, such as 10.x.x.x or 169.198.x.x in IPv4, or site-local IPv6 addresses (starting fec0 but now deprecated), or IPv6 unique local addresses (starting fd00). Nodes and pods within a Kubernetes network will normally have a private-use IPv4 address, but this can be changed.
- Addresses that are not valid for use with HTTP, such as multicast addresses or the wildcard address 0.0.0.0.

Listing 10.15 shows how to check for URLs that resolve to local or private IP addresses using Java's `java.net.InetAddress` class. This class can handle both IPv4 and IPv6 addresses and provides helper methods to check for most of the types of IP address listed previously. The only check it doesn't do is for the newer unique local addresses that were a late addition to the IPv6 standards. It is easy to check for these yourself though, by checking if the address is an instance of the `Inet6Address` class and if the first two bytes of the raw address are the values `0xFD` and `0x00`. Because the hostname in a URL may resolve to more than one IP address, you should check each address using `InetAddress.getAllByName ()`. If any address is private-use, then the code rejects the request. Open the LinkPreviewService.java file and add the two new methods from listing 10.15 to the file.

Listing 10.15 Checking for local IP addresses

```
private static boolean isBlockedAddress(String uri)
        throws UnknownHostException {
    var host = URI.create(uri).getHost();                        ❶
    for (var ipAddr : InetAddress.getAllByName(host)) {          ❷
        if (ipAddr.isLoopbackAddress() ||                        ❸
                ipAddr.isLinkLocalAddress() ||                   ❸
                ipAddr.isSiteLocalAddress() ||                   ❸
                ipAddr.isMulticastAddress() ||                   ❸
                ipAddr.isAnyLocalAddress() ||                    ❸
                isUniqueLocalAddress(ipAddr)) {                  ❸
            return true;                                         ❸
        }
    }
    return false;                                                ❹
}

private static boolean isUniqueLocalAddress(InetAddress ipAddr) {
    return ipAddr instanceof Inet6Address &&                     ❺
            (ipAddr.getAddress()[0] & 0xFF) == 0xFD &&           ❺
            (ipAddr.getAddress()[1] & 0xFF) == 0X00;             ❺
}
```

❶ Extract the hostname from the URI.

❷ Check all IP addresses for this hostname.

❸ Check if the IP address is any local- or private-use type.

❹ Otherwise, return false.

❺ To check for IPv6 unique local addresses, check the first two bytes of the raw address.

You can now update the link-preview operation to reject requests using a URL that resolves to a local address by changing the implementation of the GET request handler to reject requests for which `isBlockedAddress` returns true. Find the definition of the GET handler in the LinkPreviewService.java file and add the check as shown below in bold:

```
get("/preview", (request, response) -> {
    var url = request.queryParams("url");
    if (isBlockedAddress(url)) {
        throw new IllegalArgumentException(
                "URL refers to local/private address");
    }
```

Although this change prevents the most obvious SSRF attacks, it has some limitations:

- You’re checking only the original URL that was provided to the service, but jsoup by default will follow redirects. An attacker can set up a public website such as http://evil.example.com, which returns a HTTP redirect to an internal address inside your cluster. Because only the original URL is validated (and appears to be a genuine site), jsoup will end up following the redirect and fetching the internal site.
- Even if you allowlist a set of known good websites, an attacker may be able to find an open redirect vulnerability on one of those sites that allows them to pull off the same trick and redirect jsoup to an internal address.

**DEFINITION** An open redirect vulnerability occurs when a legitimate website can be tricked into issuing a HTTP redirect to a URL supplied by the attacker. For example, many login services (including OAuth2) accept a URL as a query parameter and redirect the user to that URL after authentication. Such parameters should always be strictly validated against a list of allowed URLs.

You can ensure that redirect URLs are validated for SSRF attacks by disabling the automatic redirect handling behavior in jsoup and implementing it yourself, as shown in listing 10.16. By calling `followRedirects(false )` the built-in behavior is prevented, and jsoup will return a response with a 3xx HTTP status code when a redirect occurs. You can then retrieve the redirected URL from the Location header on the response. By performing the URL validation inside a loop, you can ensure that all redirects are validated, not just the first URL. Make sure you define a limit on the number of redirects to prevent an infinite loop. When the request returns a non-redirect response, you can parse the document and process it as before. Open the LinkPreviewer.java file and add the method from listing 10.16.

Listing 10.16 Validating redirects

```
private static Document fetch(String url) throws IOException {
    Document doc = null;
    int retries = 0;
    while (doc == null && retries++ < 10) {                     ❶
        if (isBlockedAddress(url)) {                            ❷
            throw new IllegalArgumentException(                 ❷
                    "URL refers to local/private address");     ❷
        }                                                       ❷
        var res = Jsoup.connect(url).followRedirects(false)     ❸
                .timeout(3000).method(GET).execute();
        if (res.statusCode() / 100 == 3) {                      ❹
            url = res.header("Location");                       ❹
        } else {
            doc = res.parse();                                  ❺
        }
    }
    if (doc == null) throw new IOException("too many redirects");
    return doc;
}
```

❶ Loop until the URL resolves to a document. Set a limit on the number of redirects.

❷ If any URL resolves to a private-use IP address, then reject the request.

❸ Disable automatic redirect handling in jsoup.

❹ If the site returns a redirect status code (3xx in HTTP), then update the URL.

❺ Otherwise, parse the returned document.

Update the request handler to call the new method instead of call jsoup directly. In the handler for GET requests to the /preview endpoint, replace the line that currently reads

```
var doc = Jsoup.connect(url).timeout(3000).get();
```

with the following call to the new `fetch` "method:

```
var doc = fetch(url);
```

Pop quiz

4. Which one of the following is the most secure way to validate URLs to prevent SSRF attacks?
    1. Only performing GET requests
    2. Only performing HEAD requests
    3. Blocklisting private-use IP addresses
    4. Limiting the number of requests per second
    5. Strictly matching the URL against an allowlist of known safe values

The answer is at the end of the chapter.

### 10.2.8 DNS rebinding attacks

A more sophisticated SSRF attack, which can defeat validation of redirects, is a DNS rebinding attack, in which an attacker sets up a website and configures the DNS server for the domain to a server under their control (figure 10.5). When the validation code looks up the IP address, the DNS server returns a genuine external IP address with a very short time-to-live value to prevent the result being cached. After validation has succeeded, jsoup will perform another DNS lookup to actually connect to the website. For this second lookup, the attacker's DNS server returns an internal IP address, and so jsoup attempts to connect to the given internal service.

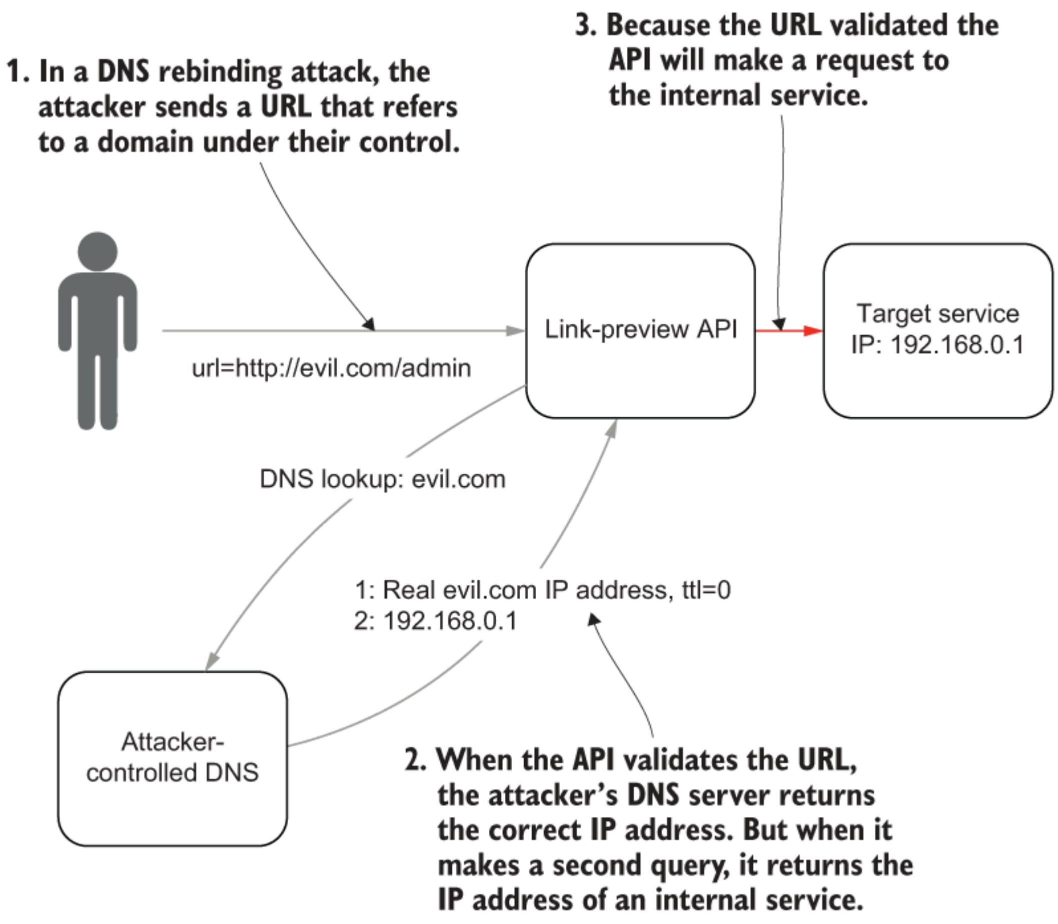
1. In a DNS rebinding attack, the attacker sends a URL that refers to a domain under their control.
3. Because the URL validated the API will make a request to the internal service.
url=http://evil.com/admin
Link-preview API
Target service IP: 192.168.0.1
DNS lookup: evil.com
1: Real evil.com IP address, ttl=0
2: 192.168.0.1
Attacker-controlled DNS
2. When the API validates the URL, the attacker's DNS server returns the correct IP address. But when it makes a second query, it returns the IP address of an internal service.

Figure 10.5 In a DNS rebinding attack, the attacker submits a URL referring to a domain under their control. When the API performs a DNS lookup during validation, the attacker's DNS server returns a legitimate IP address with a short time-to-live (ttl). Once validation has succeeded, the API performs a second DNS lookup to make the HTTP request, and the attacker's DNS server returns the internal IP address, causing the API to make an SSRF request even though it validated the URL.

**DEFINITION** A DNS rebinding attack occurs when an attacker sets up a fake website that they control the DNS for. After initially returning a correct IP address to bypass any validation steps, the attacker quickly switches the DNS settings to return the IP address of an internal service when the actual HTTP call is made.

Although it is hard to prevent DNS rebinding attacks when making an HTTP request, you can prevent such attacks against your APIs in several ways:

- Strictly validate the Host header in the request to ensure that it matches the hostname of the API being called. The Host header is set by clients based on the URL that was used in the request and will be wrong if a DNS rebinding attack occurs. Most web servers and reverse proxies provide configuration options to explicitly verify the Host header.

- By using TLS for all requests. In this case, the TLS certificate presented by the target server won't match the hostname of the original request and so the TLS authentication handshake will fail.
- Many DNS servers and firewalls can also be configured to block potential DNS binding attacks for an entire network by filtering out external DNS responses that resolve to internal IP addresses.

Listing 10.17 shows how to validate the host header in Spark Java by checking it against a set of valid values. Each service can be accessed within the same namespace using the short service name such as `natter-api-service`, or from other namespaces in the cluster using a name like `natter-api-service.natter-api`. Finally, they will also have a fully qualified name, which by default ends in `.svc.cluster.local`. Add this filter to the Natter API and the link-preview microservice to prevent attacks against those services. Open the Main.java file and add the contents of the listing to the main method, just after the existing rate-limiting filter you added in chapter 3. Add the same code to the `LinkPreviewer` class.

Listing 10.17 Validating the Host header

```
var expectedHostNames = Set.of(
        "api.natter.com",
        "api.natter.com:30567",
        "natter-link-preview-service:4567",
        "natter-link-preview-service.natter-api:4567",
        "natter-link-preview-service.natter-api.svc.cluster.local:4567");
before((request, response) -> {
    if (!expectedHostNames.contains(request.host())) {
        halt(400);
    }
});
```

❶ Define all valid hostnames for your API.

❷ Reject any request that doesn't match one of the set.

If you want to be able to call the Natter API from curl, you'll also need to add the external Minikube IP address and port, which you can get by running the command, `minikube ip`. For example, on my system I needed to add

```
"192.168.99.116:30567"
```

to the allowed host values in Main.java.

**TIP** You can create an alias for the Minikube IP address in the /etc/hosts file on Linux or MacOS by running the command `sudo sh -c "echo '$(minikube ip) api.natter.local' >> /etc/hosts`. On Windows, create or edit the file under C:\Windows\system32\etc\hosts and add a line with the IP address a space and the hostname. You can then make curl calls to http://api.natter .local:30567 rather than using the IP address.

## 10.3 Securing microservice communications

You've now deployed some APIs to Kubernetes and applied some basic security controls to the pods themselves by adding security annotations and using minimal Docker base images. These measures make it harder for an attacker to break out of a container if they find a vulnerability to exploit. But even if they can't break out from the container, they may still be able to cause a lot of damage by observing network traffic and sending their own messages on the network. For example, by observing communications between the Natter API and the H2 database they can capture the connection password and then use this to directly connect to the database, bypassing the API. In this section, you'll see how to enable additional network protections to mitigate against these attacks.

### 10.3.1 Securing communications with TLS

In a traditional network, you can limit the ability of an attacker to sniff network communications by using network segmentation. Kubernetes clusters are highly dynamic, with pods and services coming and going as configuration changes, but low-level network segmentation is a more static approach that is hard to change. For this reason, there is usually no network segmentation of this kind within a Kubernetes cluster (although there might be between clusters running on the same infrastructure), allowing an attacker that gains privileged access to observe all network communications within the cluster by default. They can use credentials discovered from this snooping to access other systems and increase the scope of the attack.

**DEFINITION** Network segmentation refers to using switches, routers, and firewalls to divide a network into separate segments (also known as collision domains). An attacker can then only observe network traffic within the same network segment and not traffic in other segments.

Although there are approaches that provide some of the benefits of segmentation within a cluster, a better approach is to actively protect all communications using TLS. Apart from preventing an attacker from snooping on network traffic, TLS also protects against a range of attacks at the network level, such as the DNS rebind attacks mentioned in section 10.2.8. The certificate-based authentication built into TLS protects against spoofing attacks such as DNS cache poisoning or ARP spoofing, which rely on the lack of authentication in low-level protocols. These attacks are prevented by firewalls, but if an attacker is inside your network (behind the firewall) then they can often be carried out effectively. Enabling TLS inside your cluster significantly reduces the ability of an attacker to expand an attack after gaining an initial foothold.

**DEFINITION** In a DNS cache poisoning attack, the attacker sends a fake DNS message to a DNS server changing the IP address that a hostname resolves to. An ARP spoofing attack works at a lower level by changing the hardware address (ethernet MAC address, for example) that an IP address resolves to.

To enable TLS, you'll need to generate certificates for each service and distribute the certificates and private keys to each pod that implements that service. The processes involved in creating and distributing certificates is known as public key infrastructure (PKI).

**DEFINITION** A public key infrastructure is a set of procedures and processes for creating, distributing, managing, and revoking certificates used to authenticate TLS connections.

Running a PKI is complex and error-prone because there are a lot of tasks to consider:

- Private keys and certificates have to be distributed to every service in the network and kept secure.
- Certificates need to be issued by a private certificate authority (CA), which itself needs to be secured. In some cases, you may want to have a hierarchy of CAs with a root CA and one or more intermediate CAs for additional security. Services which are available to the public must obtain a certificate from a public CA.
- Servers must be configured to present a correct certificate chain and clients must be configured to trust your root CA.

- Certificates must be revoked when a service is decommissioned or if you suspect a private key has been compromised. Certificate revocation is done by publishing and distributing certificate revocation lists (CRLs) or running an online certificate status protocol (OCSP) service.
- Certificates must be automatically renewed periodically to prevent them from expiring. Because revocation involves blocklisting a certificate until it expires, short expiry times are preferred to prevent CRLs becoming too large. Ideally, certificate renewal should be completely automated.

Using an intermediate CA

Directly issuing certificates from the root CA trusted by all your microservices is simple, but in a production environment, you'll want to automate issuing certificates. This means that the CA needs to be an online service responding to requests for new certificates. Any online service can potentially be compromised, and if this is the root of trust for all TLS certificates in your cluster (or many clusters), then you'd have no choice in this case but to rebuild the cluster from scratch. To improve the security of your clusters, you can instead keep your root CA keys offline and only use them to periodically sign an intermediate CA certificate. This intermediate CA is then used to issue certificates to individual microservices. If the intermediate CA is ever compromised, you can use the root CA to revoke its certificate and issue a new one. The root CA certificate can then be very long-lived, while intermediate CA certificates are changed regularly.

To get this to work, each service in the cluster must be configured to send the intermediate CA certificate to the client along with its own certificate, so that the client can construct a valid certificate chain from the service certificate back to the trusted root CA.

If you need to run multiple clusters, you can also use a separate intermediate CA for each cluster and use name constraints (**http://mng.bz/oR8r**) in the intermediate CA certificate to restrict which names it can issue certificates for (but not all clients support name constraints). Sharing a common root CA allows clusters to communicate with each other easily, while the separate intermediate CAs reduce the scope if a compromise occurs.

### 10.3.2 Using a service mesh for TLS

In a highly dynamic environment like Kubernetes, it is not advisable to attempt to run a PKI manually. There are a variety of tools available to help run a PKI for you. For example, Cloudflare's PKI toolkit (**https://cfssl.org**) and Hashicorp Vault (**http:// mng.bz/nzrg**) can both be used to automate most aspects of running a PKI. These general-purpose tools still require a significant amount of effort to integrate into a Kubernetes environment. An alternative that is becoming more popular in recent years is to use a service mesh such as Istio (**https://istio.io**) or Linkerd (**https://linkerd.io**) to handle TLS between services in your cluster for you.

**DEFINITION** A service mesh is a set of components that secure communications between pods in a cluster using proxy sidecar containers. In addition to security benefits, a service mesh provides other useful functions such as load balancing, monitoring, logging, and automatic request retries.

A service mesh works by installing lightweight proxies as sidecar containers into every pod in your network, as shown in figure 10.6. These proxies intercept all network requests coming into the pod (acting as a reverse proxy) and all requests going out of the pod. Because all communications flow through the proxies, they can transparently initiate and terminate TLS, ensuring that communications across the network are secure while the individual microservices use normal unencrypted messages. For example, a client can make a normal HTTP request to a REST API and the client's service mesh proxy (running inside the same pod on the same machine) will transparently upgrade this to HTTPS. The proxy at the receiver will handle the TLS connection and forward the plain HTTP request to the target service. To make this work, the service mesh runs a central CA service that distributes certificates to the proxies. Because the service mesh is aware of Kubernetes service metadata, it automatically generates correct certificates for each service and can periodically reissue them.[2]

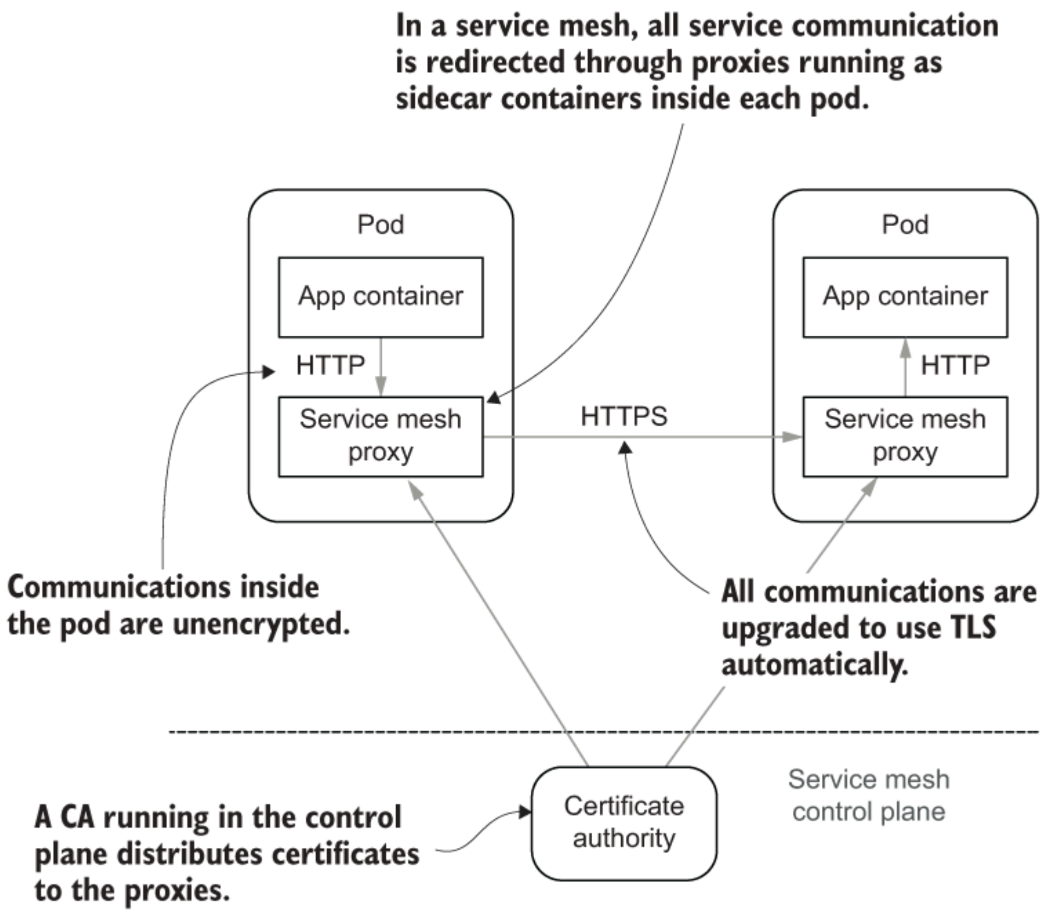
In a service mesh, all service communication is redirected through proxies running as sidecar containers inside each pod.
Pod
App container
HTTP
Service mesh proxy
HTTPS
Pod
App container
HTTP
Service mesh proxy
Communications inside the pod are unencrypted.
All communications are upgraded to use TLS automatically.
A CA running in the control plane distributes certificates to the proxies.
Certificate authority
Service mesh control plane

Figure 10.6 In a service mesh, a proxy is injected into each pod as a sidecar container. All requests to and from the other containers in the pod are redirected through the proxy. The proxy upgrades communications to use TLS using certificates it obtains from a CA running in the service mesh control plane.

To enable a service mesh, you need to install the service mesh control plane components such as the CA into your cluster. Typically, these will run in their own Kubernetes namespace. In many cases, enabling TLS is then simply a case of adding some annotations to the deployment YAML files. The service mesh will then automatically inject the proxy sidecar container when your pods are started and configure them with TLS certificates.

In this section, you'll install the Linkerd service mesh and enable TLS between the Natter API, its database, and the link-preview service, so that all communications are secured within the network. Linkerd has fewer features than Istio, but is much simpler to deploy and configure, which is why I've chosen it for the examples in this book. From a security perspective, the relative simplicity of Linkerd reduces the opportunity for vulnerabilities to be introduced into your cluster.

**DEFINITION** The control plane of a service mesh is the set of components responsible for configuring, managing, and monitoring the proxies. The proxies themselves and the services they protect are known as the data plane.

*INSTALLING LINKERD*

To install Linkerd, you first need to install the `linkerd` command-line interface (CLI), which will be used to configure and control the service mesh. If you have Homebrew installed on a Mac or Linux box, then you can simply run the following command:

```
brew install linkerd
```

On other platforms it can be downloaded and installed from [https://github.com/ linkerd/linkerd2/releases/](https://github.com/linkerd/linkerd2/releases/). Once you've installed the CLI, you can run pre-installation checks to ensure that your Kubernetes cluster is suitable for running the service mesh by running:

```
linkerd check --pre
```

If you've followed the instructions for installing Minikube in this chapter, then this will all succeed. You can then install the control plane components by running the following command:

```
linkerd install | kubectl apply -f -
```

Finally, run `linkerd` `check` again (without the `--pre` argument) to check the progress of the installation and see when all the components are up and running. This may take a few minutes as it downloads the container images.

To enable the service mesh for the Natter namespace, edit the namespace YAML file to add the `linkerd` annotation, as shown in listing 10.18. This single annotation will ensure that all pods in the namespace have Linkerd sidecar proxies injected the next time they are restarted.

Listing 10.18 Enabling Linkerd

```
apiVersion: v1
kind: Namespace
metadata:
  name: natter-api
```

```
    labels:
      name: natter-api
    annotations:                        (1)
      linkerd.io/inject: enabled        (1)
```

(1) Add the linkerd annotation to enable the service mesh.

Run the following command to update the namespace definition:

```
kubectl apply -f kubernetes/natter-namespace.yaml
```

You can force a restart of each deployment in the namespace by running the following commands:

```
kubectl rollout restart deployment \
  natter-database-deployment -n natter-api
kubectl rollout restart deployment \
  link-preview-deployment -n natter-api
kubectl rollout restart deployment \
  natter-api-deployment -n natter-api
```

For HTTP APIs, such as the Natter API itself and the link-preview microservice, this is all that is required to upgrade those services to HTTPS when called from other services within the service mesh. You can verify this by using the Linkerd `tap` utility, which allows for monitoring network connections in the cluster. You can start tap by running the following command in a new terminal window:

```
linkerd tap ns/natter-api
```

If you then request a message that contains a link to trigger a call to the link-preview service (using the steps at the end of section 10.2.6), you'll see the network requests in the tap output. This shows the initial request from curl without TLS (`tls` `=` `not_provided _by_remote`), followed by the request to the link-preview service with TLS enabled (`tls` `=` `true`). Finally, the response is returned to curl without TLS:

```
req id=2:0 proxy=in  src=172.17.0.1:57757 dst=172.17.0.4:4567     (1)
➥ tls=not_provided_by_remote :method=GET :authority=              (1)
➥ natter-api-service:4567 :path=/spaces/1/messages/1              (1)
req id=2:1 proxy=out src=172.17.0.4:53996 dst=172.17.0.16:4567    (2)
```

```
  ➥ tls=true :method=GET :authority=natter-link-preview-      ❷
  ➥ service:4567 :path=/preview                                ❷
rsp id=2:1 proxy=out src=172.17.0.4:53996 dst=172.17.0.16:4567  ❷
  ➥ tls=true :status=200 latency=479094µs                      ❷
end id=2:1 proxy=out src=172.17.0.4:53996 dst=172.17.0.16:4567  ❷
  ➥ tls=true duration=665µs response-length=330B               ❷
rsp id=2:0 proxy=in  src=172.17.0.1:57757 dst=172.17.0.4:4567   ❷
  ➥ tls=not_provided_by_remote :status=200 latency=518314µs    ❸
end id=2:0 proxy=in  src=172.17.0.1:57757                       ❸
  ➥ dst=172.17.0.4:4567 tls=not_provided_by_remote duration=169µs ❸
  ➥ response-length=428B                                       ❸
```

❶ The initial response from curl is not using TLS.

❷ The internal call to the link-preview service is upgraded to TLS.

❸ The response back to curl is also sent without TLS.

You'll enable TLS for requests coming into the network from external clients in section 10.4.

Mutual TLS

Linkerd and most other service meshes don't just supply normal TLS server certificates, but also client certificates that are used to authenticate the client to the server. When both sides of a connection authenticate using certificates this is known as mutual TLS, or mutually authenticated TLS, often abbreviated mTLS. It's important to know that mTLS is not by itself any more secure than normal TLS. There are no attacks against TLS at the transport layer that are prevented by using mTLS. The purpose of a server certificate is to prevent the client connecting to a fake server, and it does this by authenticating the hostname of the server. If you recall the discussion of authentication in chapter 3, the server is claiming to be `api.example.com` and the server certificate authenticates this claim. Because the server does not initiate connections to the client, it does not need to authenticate anything for the connection to be secure.

The value of mTLS comes from the ability to use the strongly authenticated client identity communicated by the client certificate to enforce API authorization policies at the server. Client certificate authenticate is significantly more secure than many other authentication mechanisms but is complex to configure and maintain. By handling this for you, a service mesh enables strong API authentication mechanisms. In chapter 11, you'll

learn how to combine mTLS with OAuth2 to combine strong client authentication with token-based authorization.
The current version of Linkerd can automatically upgrade only HTTP traffic to use TLS, because it relies on reading the HTTP Host header to determine the target service. For other protocols, such as the protocol used by the H2 database, you'd need to manually set up TLS certificates.

**TIP** Some service meshes, such as Istio, can automatically apply TLS to non-HTTP traffic too.[3] This is planned for the 2.7 release of Linkerd. See Istio in Action by Christian E. Posta (Manning, 2020) if you want to learn more about Istio and service meshes in general.

Pop quiz

5. Which of the following are reasons to use an intermediate CA? Select all that apply.
   1. To have longer certificate chains
   2. To keep your operations teams busy
   3. To use smaller key sizes, which are faster
   4. So that the root CA key can be kept offline
   5. To allow revocation in case the CA key is compromised
6. True or False: A service mesh can automatically upgrade network requests to use TLS.

The answers are at the end of the chapter.

### 10.3.3 Locking down network connections

Enabling TLS in the cluster ensures that an attacker can't modify or eavesdrop on communications between APIs in your network. But they can still make their own connections to any service in any namespace in the cluster. For example, if they compromise an application running in a separate namespace, they can make direct connections to the H2 database running in the `natter-api` namespace. This might allow them to attempt to guess the connection password, or to scan services in the network for vulnerabilities to exploit. If they find a vulnerability, they can then compromise that service and find new attack possibilities. This process of moving from service to service inside your network after an initial compromise is known as lateral movement and is a common tactic.

**DEFINITION** Lateral movement is the process of an attacker moving from system to system within your network after an initial compromise. Each new system compromised provides new opportunities to carry out further attacks, expanding the systems under the attacker's control. You can learn more about common attack tactics through frameworks such as MITRE ATT&CK (https://attack .mitre.org).

To make it harder for an attacker to carry out lateral movement, you can apply network policies in Kubernetes that restrict which pods can connect to which other pods in a network. A network policy allows you to state which pods are expected to connect to each other and Kubernetes will then enforce these rules to prevent access from other pods. You can define both ingress rules that determine what network traffic is allowed into a pod, and egress rules that say which destinations a pod can make outgoing connections to.

**DEFINITION** A Kubernetes network policy (http://mng.bz/v94J) defines what network traffic is allowed into and out of a set of pods. Traffic coming into a pod is known as ingress, while outgoing traffic from the pod to other hosts is known as egress.

Because Minikube does not support network policies currently, you won't be able to apply and test any network policies created in this chapter. Listing 10.19 shows an example network policy that you could use to lock down network connections to and from the H2 database pod. Apart from the usual name and namespace declarations, a network policy consists of the following parts:

- A `podSelector` that describes which pods in the namespace the policy will apply to. If no policies select a pod, then it will be allowed all ingress and egress traffic by default, but if any do then it is only allowed traffic that matches at least one of the rules defined. The `podSelector: {}` syntax can be used to select all pods in the namespace.
- A set of policy types defined in this policy, out of the possible values `Ingress` and `Egress`. If only ingress policies are applicable to a pod then Kubernetes will still permit all egress traffic from that pod by default, and vice versa. It's best to explicitly define both Ingress and Egress policy types for all pods in a namespace to avoid confusion.
- An `ingress` section that defines allowlist ingress rules. Each ingress rule has a `from` section that says which other pods, namespaces, or IP address ranges can make network connections to the pods in this policy. It also has a `ports` section that defines which TCP and UDP ports those clients can connect to.
- An `egress` section that defines the allowlist egress rules. Like the ingress rules, egress rules consist of a `to` section defining the allowed destinations and a `ports` section defining the allowed target ports.

**TIP** Network policies apply to only new connections being established. If an incoming connection is permitted by the ingress policy rules, then any outgoing traffic related to that connection will be permitted without defining individual egress rules for each possible client.

Listing 10.19 defines a complete network policy for the H2 database. For ingress, it defines a rule that allows connections to TCP port 9092 from pods with the label `app: natter-api`. This allows the main Natter API pods to talk to the database. Because no other ingress rules are defined, no other incoming connections will be accepted. The policy in listing 10.19 also lists the `Egress` policy type but doesn't define any egress rules, which means that all outbound connections from the database pods will be blocked. This listing is to illustrate how network policies work; you don't need to save the file anywhere.

**NOTE** The allowed ingress or egress traffic is the union of all policies that select a pod. For example, if you add a second policy that permits the database pods to make egress connections to google.com then this will be allowed even though the first policy doesn't allow this. You must examine all policies in a namespace together to determine what is allowed.

Listing 10.19 Token database network policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-network-policy
  namespace: natter-api
spec:
  podSelector:                    ❶
    matchLabels:                  ❶
      app: natter-database        ❶
  policyTypes:
    - Ingress                     ❷
    - Egress                      ❷
  ingress:
    - from:                       ❸
        - podSelector:            ❸
            matchLabels:          ❸
              app: natter-api     ❸
      ports:                      ❹
        - protocol: TCP           ❹
          port: 9092              ❹
```

❶ Apply the policy to pods with the app=natter-database label.

❷ The policy applies to both incoming (ingress) and outgoing (egress) traffic.

❸ Allow ingress only from pods with the label app=natter-api-service in the same namespace.

❹ Allow ingress only to TCP port 9092.

You can create the policy and apply it to the cluster using `kubectl apply`, but on Minikube it will have no effect because Minikube's default networking components are not able to enforce policies. Most hosted Kubernetes services, such as those provided by Google, Amazon, and Microsoft, do support enforcing network policies. Consult the documentation for your cloud provider to see how to enable this. For self-hosted Kubernetes clusters, you can install a network plugin such as Calico (**https://www.projectcalico.org**) or Cilium (**https://cilium.readthedocs.io/en/v1.6/**).

As an alternative to network policies, Istio supports defining network authorization rules in terms of the service identities contained in the client certificates it uses for mTLS within the service mesh. These policies go beyond what is supported by network policies and can control access based on HTTP methods and paths. For example, you can allow one service to only make GET requests to another service. See **http://mng.bz/4BKa** for more details. If you have a dedicated security team, then service mesh authorization allows them to enforce consistent security controls across the cluster, allowing API development teams to concentrate on their unique security requirements.

**WARNING** Although service mesh authorization policies can significantly harden your network, they are not a replacement for API authorization mechanisms. For example, service mesh authorization provides little protection against the SSRF attacks discussed in section 10.2.7 because the malicious requests will be transparently authenticated by the proxies just like legitimate requests.

## 10.4 Securing incoming requests

So far, you've only secured communications between microservice APIs within the cluster. The Natter API can also be called by clients outside the cluster, which you've been doing with curl. To secure requests into the cluster, you can enable an ingress controller that will receive all requests arriving from external sources as shown in figure 10.7. An ingress controller is a reverse proxy or load balancer, and can be configured to perform TLS termination, rate-limiting, audit logging, and other basic security controls. Requests that pass these checks are then forwarded on to the services within the network. Because the ingress controller itself runs within the network, it can be included in the Linkerd service mesh, ensuring that the forwarded requests are automatically upgraded to HTTPS.

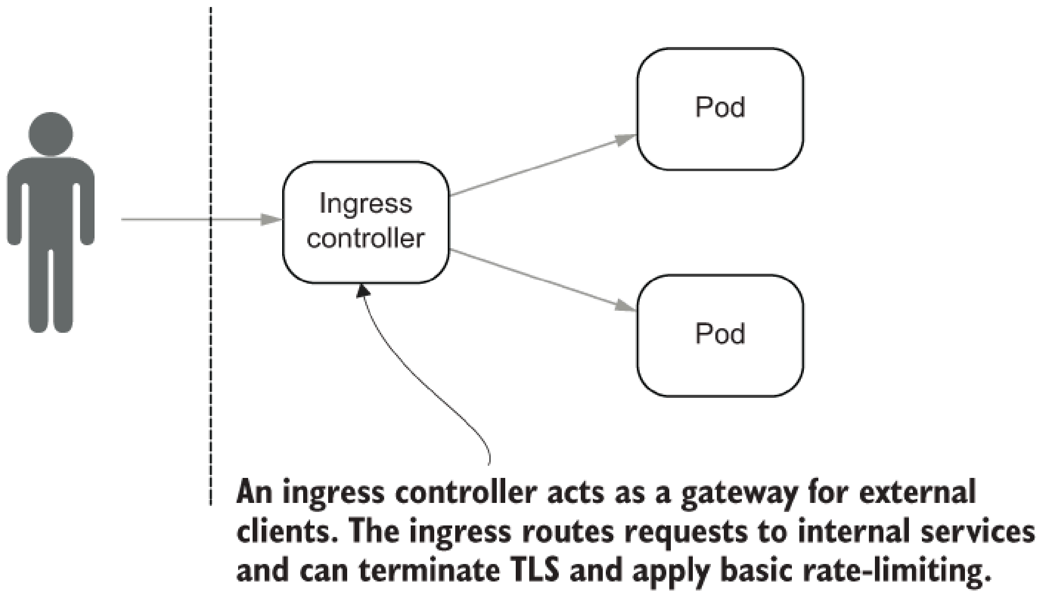
Pod
Ingress controller
Pod
An ingress controller acts as a gateway for external clients. The ingress routes requests to internal services and can terminate TLS and apply basic rate-limiting.

Figure 10.7 An ingress controller acts as a gateway for all requests from external clients. The ingress can perform tasks of a reverse proxy or load balancer, such as terminating TLS connections, performing rate-limiting, and adding audit logging.

**DEFINITION** A Kubernetes ingress controller is a reverse proxy or load balancer that handles requests coming into the network from external clients. An ingress controller also often functions as an API gateway, providing a unified API for multiple services within the cluster.

**NOTE** An ingress controller usually handles incoming requests for an entire Kubernetes cluster. Enabling or disabling an ingress controller may therefore have implications for all pods running in all namespaces in that cluster.

To enable an ingress controller in Minikube, you need to enable the `ingress` add-on. Before you do that, if you want to enable mTLS between the ingress and your services you can annotate the kube-system namespace to ensure that the new ingress pod that gets created will be part of the Linkerd service mesh. Run the following two commands to launch the ingress controller inside the service mesh. First run

```
kubectl annotate namespace kube-system linkerd.io/inject=enabled
```

and then run:

```
minikube addons enable ingress
```

This will start a pod within the kube-system namespace running the NGINX web server (https://nginx.org), which is configured to act as a reverse proxy. The ingress controller will take a few minutes to start. You can check its progress by running the command:

```
kubectl get pods -n kube-system --watch
```

After you have enabled the ingress controller, you need to tell it how to route requests to the services in your namespace. This is done by creating a new YAML configuration file with kind `Ingress`. This configuration file can define how HTTP requests are mapped to services within the namespace, and you can also enable TLS, rate-limiting, and other features (see http://mng.bz/Qxqw for a list of features that can be enabled).

Listing 10.20 shows the configuration for the Natter ingress controller. To allow Linkerd to automatically apply mTLS to connections between the ingress controller and the backend services, you need to rewrite the Host header from the external value (such as api.natter.local) to the internal name used by your service. This can be achieved by adding the `nginx.ingress.kubernetes.io/upstream-vhost` annotation. The NGINX configuration defines variables for the service name, port, and namespace based on the configuration so you can use these in the definition. Create a new file named natter-ingress.yaml in the kubernetes folder with the contents of the listing, but don't apply it just yet. There's one more step you need before you can enable TLS.

**TIP** If you're not using a service mesh, your ingress controller may support establishing its own TLS connections to backend services or proxying TLS connections straight through to those services (known as SSL passthrough). Istio includes an alternative ingress controller, Istio Gateway, that knows how to connect to the service mesh.

Listing 10.20 Configuring ingress

```
apiVersion: extensions/v1beta1
kind: Ingress                                                          ❶
metadata:
  name: api-ingress                                                    ❷
  namespace: natter-api                                                ❷
  annotations:
    nginx.ingress.kubernetes.io/upstream-vhost:                        ❸
      "$service_name.$namespace.svc.cluster.local:$service_port"       ❸
spec:
  tls:                                                                 ❹
    - hosts:                                                           ❹
        - api.natter.local                                             ❹
      secretName: natter-tls                                           ❹
  rules:                                                               ❺
    - host: api.natter.local                                           ❺
      http:                                                            ❺
        paths:                                                         ❺
          - backend:                                                   ❺
              serviceName: natter-api-service                          ❺
              servicePort: 4567                                        ❺
```

❶ Define the Ingress resource.

❷ Give the ingress rules a name in the natter-api namespace.

❸ Rewrite the Host header using the upstream-vhost annotation.

❹ Enable TLS by providing a certificate and key.

❺ Define a route to direct all HTTP requests to the natter-api-service.

To allow the ingress controller to terminate TLS requests from external clients, it needs to be configured with a TLS certificate and private key. For development, you can create a certificate with the `mkcert` utility that you used in chapter 3:

```
mkcert api.natter.local
```

This will spit out a certificate and private key in the current directory as two files with the `.pem` extension. PEM stands for Privacy Enhanced Mail and is a common file format for keys and certificates. This is also the format that the ingress controller needs. To make the key and certificate available to the ingress, you need to create a Kubernetes secret to hold them.

**DEFINITION** Kubernetes secrets are a standard mechanism for distributing passwords, keys, and other credentials to pods running in a cluster. The secrets are stored in a central database and distributed to pods as either filesystem mounts or environment variables. You'll learn more about Kubernetes secrets in chapter 11.

To make the certificate available to the ingress, run the following command:

```
kubectl create secret tls natter-tls -n natter-api \
  --key=api.natter.local-key.pem --cert=api.natter.local.pem
```

This will create a TLS secret with the name `natter-tls` in the `natter-api` namespace with the given key and certificate files. The ingress controller will be able to find this secret because of the `secretName` configuration option in the ingress configuration file. You can now create the ingress configuration to expose the Natter API to external clients:

```
kubectl apply -f kubernetes/natter-ingress.yaml
```

You'll now be able to make direct HTTPS calls to the API:

```
$ curl https://api.natter.local/users \
  -H 'Content-Type: application/json' \
  -d '{"username":"abcde","password":"password"}'
{"username":"abcde"}
```

If you check the status of requests using Linkerd's `tap` utility, you'll see that requests from the ingress controller are protected with mTLS:

```
$ linkerd tap ns/natter-api
req id=4:2 proxy=in  src=172.17.0.16:43358 dst=172.17.0.14:4567
➥ tls=true :method=POST :authority=natter-api-service.natter-
➥ api.svc.cluster.local:4567 :path=/users
rsp id=4:2 proxy=in  src=172.17.0.16:43358 dst=172.17.0.14:4567
➥ tls=true :status=201 latency=322728µs
```

You now have TLS from clients to the ingress controller and mTLS between the ingress controller and backend services, and between all microservices on the backend.[4]

**TIP** In a production system you can use cert-manager (**https://docs.cert-manager.io/en/latest/**) to automatically obtain certificates from a public CA such as Let's Encrypt or from a private organizational CA such as Hashicorp Vault.

Pop quiz

7. Which of the following are tasks are typically performed by an ingress controller?
   1. Rate-limiting
   2. Audit logging
   3. Load balancing
   4. Terminating TLS requests
   5. Implementing business logic
   6. Securing database connections

The answer is at the end of the chapter.

## Answers to pop quiz questions

1. c. Pods are made up of one or more containers.
2. False. A sidecar container runs alongside the main container. An init container is the name for a container that runs before the main container.
3. a, b, c, d, and f are all good ways to improve the security of containers.
4. e. You should prefer strict allowlisting of URLs whenever possible.
5. d and e. Keeping the root CA key offline reduces the risk of compromise and allows you to revoke and rotate intermediate CA keys without rebuilding the whole cluster.
6. True. A service mesh can automatically handle most aspects of applying TLS to your network requests.
7. a, b, c, and d.

## Summary

- Kubernetes is a popular way to manage a collection of microservices running on a shared cluster. Microservices are deployed as pods, which are groups of related Linux containers. Pods are scheduled across nodes, which are physical or virtual machines that make up the cluster. A service is implemented by one or more pod replicas.
- A security context can be applied to pod deployments to ensure that the container runs as a non-root user with limited privileges. A pod security policy can be applied to the cluster to enforce that no container is allowed elevated privileges.
- When an API makes network requests to a URL provided by a user, you should ensure that you validate the URL to prevent SSRF attacks. Strict allowlisting of permitted URLs should be preferred to blocklisting. Ensure that redirects are also validated. Protect your APIs from DNS rebinding attacks by strictly validating the Host header and enabling TLS.
- Enabling TLS for all internal service communications protects against a variety of attacks and limits the damage if an attacker breaches your network. A service mesh such as Linkerd or Istio can be used to automatically manage mTLS connections between all services.
- Kubernetes network policies can be used to lock down allowed network communications, making it harder for an attacker to perform lateral movement inside your network. Istio authorization policies can perform the same task based on service identities and may be easier to configure.
- A Kubernetes ingress controller can be used to allow connections from external clients and apply consistent TLS and rate-limiting options. By adding the ingress controller to the service mesh you can ensure connections from the ingress to backend services are also protected with mTLS.

---

[1]. Restarting Minikube will also delete the contents of the database as it is still purely in-memory. See http://mng.bz/5pZ1 for details on how to enable persistent disk volumes that survive restarts.

[2]. At the time of writing, most service meshes don't support certificate revocation, so you should use short-lived certificates and avoid relying on this as your only authentication mechanism.

[3]. Istio has more features that Linkerd but is also more complex to install and configure, which is why I chose Linkerd for this chapter.

4. The exception is the H2 database as Linkerd can't automatically apply mTLS to this connection. This should be fixed in the 2.7 release of Linkerd.