

Microservices Security IN ACTION

Prabath Siriwardena
AND Nuwan Dias



MEAP



MANNING



MEAP Edition
Manning Early Access Program
Microservices Security in Action
Version 7

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to

www.manning.com

welcome

Thanks for purchasing the MEAP of *Microservices Security in Action*. We hope that what you'll get access to will be of immediate use to you and, with your help, the final book will be great!

This book is written for developers and architects who have some experience in software development and who are well versed in microservices design principles, applications, and benefits.

In this book, we focus on microservices security. When you make the decision to use microservices architecture in building all your critical business operations, security is of topmost importance. A security breach could result in many unpleasant outcomes, from losing customer confidence to bankruptcy. Microservices are becoming key enablers of digital transformation, so microservices security must be consciously planned, designed, and implemented.

This book introduces you to the key fundamentals, security principles, and best practices involved in securing microservices. We'll be using industry-leading open-source tools along with Java code samples developed with Spring Boot for demonstrations. You may pick better competitive tools later in your production environment, of course.

The book is divided into five parts. Part 1 will fly through the challenges in securing microservices, key security fundamentals and then get your hands wet with a basic microservice, and show you how to secure it. Part 2 of the book focuses on edge security. There we teach you how to deploy microservices behind an API gateway, secure them, and access microservices via an API gateway from a single-page application (SPA). Also we discuss how to do throttling, gather analytics, and do access control at the API gateway. Part 3 talks about various options in securing microservice to microservice communication. Part 4 of the book focuses on secure deployment. There we talk about deploying microservices securely in a containerized environment with Docker and Kubernetes. Further we teach you how to secure microservices with Istio service mesh. In Part 5 we discuss the security best practices and processes you need to follow while writing microservices.

Your feedback on [liveBook Discussion Forum](#) will be invaluable in improving *Microservices Security in Action*.

Thanks again for your interest and for purchasing the MEAP!

—Prabath Siriwardena / Nuwan Dias

brief contents

PART 1 OVERVIEW

- 1 Microservices security landscape*
- 2 First steps in securing microservices*

PART 2 EDGE SECURITY

- 3 Securing north/south traffic with an API gateway*
- 4 Accessing a secured microservice via a single-page application*
- 5 Engaging throttling, monitoring, and access control*

PART 3 SERVICE-TO-SERVICE COMMUNICATION

- 6 Securing east/west traffic with certificates*
- 7 Securing east/west traffic with JWT*
- 8 Securing east/west traffic over gRPC*
- 9 Securing reactive microservices*

PART 4 SECURE DEPLOYMENT

- 10 Conquering container security with Docker*
- 11 Securing microservices on Kubernetes*
- 12 Securing microservices with Istio service mesh*

PART 5 SECURE DEVELOPMENT

- 13 Secure coding practices and automation*

APPENDICES

- A Docker fundamentals*
- B Kubernetes fundamentals*

- C Service mesh and Istio fundamentals*
- D OAuth 2.0 and OpenID Connect*
- E Single-page Application architecture*
- F Observability in a microservices deployment*
- G Open Policy Agent*
- H JSON Web Token (JWT)*
- I Secure Production Identity Framework For Everyone*
- J gRPC fundamentals*
- K Creating a certificate authority and related keys with OpenSSL*

1

Microservices security landscape

This chapter covers

- Why microservices security is challenging
- The principles and key elements of a microservices security design
- Edge security and the role of an API gateway in securing a microservices deployment
- Patterns and practices in securing service-to-service communication in a microservices deployment

Fail fast; fail often is a mantra in Silicon Valley. Not everyone agrees, but we love it! It's an invitation to experiment with new things, accept failures, fix problems, and try again. Not everything in ink looks pretty in practice. Fail fast, fail often is hype only unless the organizational leadership, the culture, and the technology are there and thrive. We find microservices to be a key enabler for fail fast, fail often. Microservices architecture has gone beyond technology and architectural principles to become a culture. Netflix, Amazon, Lyft, Uber, and eBay are the front-runners in building a culture behind microservices. Neither the architecture nor the technology behind microservices but the discipline practiced in an organizational culture lets your team build stable products, deploy them in a production environment with less hassle, and introduce frequent changes with no negative impact on the overall system. Speed to production and evolvability are the two key outcomes of microservices architecture. International Data Corporation (IDC) in its predictions for 2019 and beyond announced that by 2022, 90% of all apps would feature microservices architectures that improve the ability to design, debug, update, and leverage third-party code¹.

¹ International Data Corporation (IDC) predictions for 2019 and beyond, <https://www.idc.com/getdoc.jsp?containerId=prUS44417618>

We assume that you're well versed in microservices design principles, applications, and benefits. If you're new to microservices and have never been (or are only slightly) involved in development projects, we recommend that you read a book on microservices first, such as *Spring Microservices in Action* by John Carnell (Manning Publications, 2017). *Microservices Patterns* by Chris Richardson (Manning Publications, 2018) and *Microservices in Action* (Manning Publications, 2018) by Morgan Bruce and Paulo A. Pereira are two other good books on the subject. *Microservices for the Enterprise: Designing, Developing, and Deploying* by Prabath Siriwardena (a co-author of this book) and Kasun Indrasiri (Apress, 2018) is another beginner's book on microservices.

In this book, we focus on microservices security. When you make the decision to go ahead with microservices architecture to build all your critical business operations, security is of topmost importance. A security breach could result in many unpleasant outcomes, from losing customer confidence to bankruptcy. Emphasis on security today is higher than at any time in the past. Microservices are becoming key enablers of digital transformation, so microservices security must be consciously planned, designed, and implemented.

This book introduces you to the key fundamentals, security principles, and best practices involved in securing microservices. We'll be using industry-leading open-source tools along with Java code samples developed with Spring Boot for demonstrations. You may pick better competitive tools later in your production environment, of course.

This book will give you good understanding of how to implement microservices security concepts in real life. Even if you're not a Java developer, if you're familiar with any object-oriented programming language (such as C++ or C#) and understand basic programming constructs well, you'll still enjoy reading the book, even though its samples are in Java. Then again, security is a broader topic. It's a discipline with multiple sub-disciplines. In this book we mostly focus on application developers and architects who worry about managing access to their microservices. Access management itself is another broader sub-discipline of the larger security discipline. We will not focus on pen testing, developing threat models, firewalls, system level configurations to harden security and so on.

1.1 How security works in a monolithic application

A monolithic application has few entry points. An entry point for an application is analogous to a door in a building. As a door lets you into the building (possibly after security screening), an application entry point lets your requests in. Think about a web application (see figure 1.1) running on the default HTTP port 80 on a server carrying the IP address 192.168.0.1. Port 80 on server 192.168.0.1 is an entry point to that web application. If the same web application

accepts HTTPS requests on the same server on port 443, you have another entry point. When you have more entry points, you've got more places to worry about securing. (You need to deploy more soldiers when you have a longer border to protect, for example, or to build a wall that closes all entry points.) The higher the entry points to an application are, the broader the attack surface is.

Most monolithic applications have only a couple of entry points. Not every component of a monolithic application is exposed to the outside world and accepts requests directly.

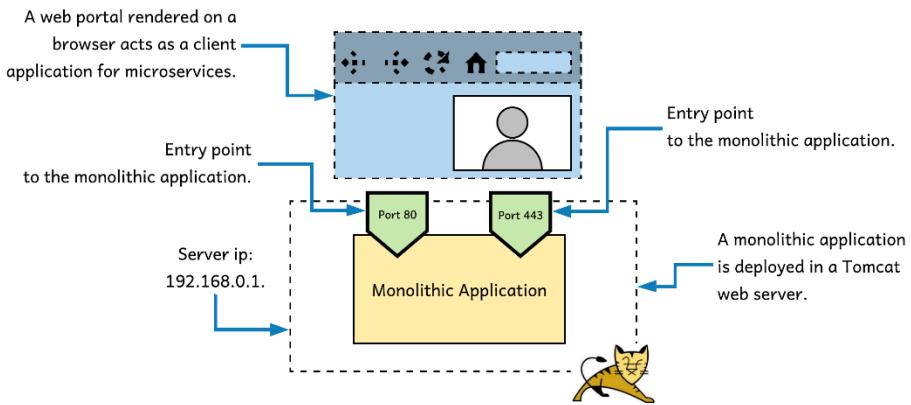


Figure 1.1 A monolithic application typically has few entry points. Here, there are two: ports 80 and 443

In a typical Java EE web application such as the one in figure 1.1, all requests are scanned for security at the application level by a servlet filter.² This security screening checks whether the current request is associated with a valid web session and, if not, challenges the requesting party to authenticate first. Further access-control checks may validate that the requesting party has the necessary permissions to do what he or she intends to do. The servlet filter (the interceptor) carries out such checks centrally to make sure that only legitimate requests are dispatched to the corresponding components. Internal components need not worry about the legitimacy of the requests; they can rightly assume that if a request lands there, all the security checks have already been done. In case those components need to know who the requesting party (or user) is or to find other information related to the user, such information can be retrieved from the web session, which is shared among all the components (see figure 1.2). The

² If you aren't familiar with servlet filters, think about them as interceptors running in the same process with the web application, intercepting all the requests to the web application.

Servlet filter injects the requesting-party information into the web session during the initial screening process after completing authentication and authorization.

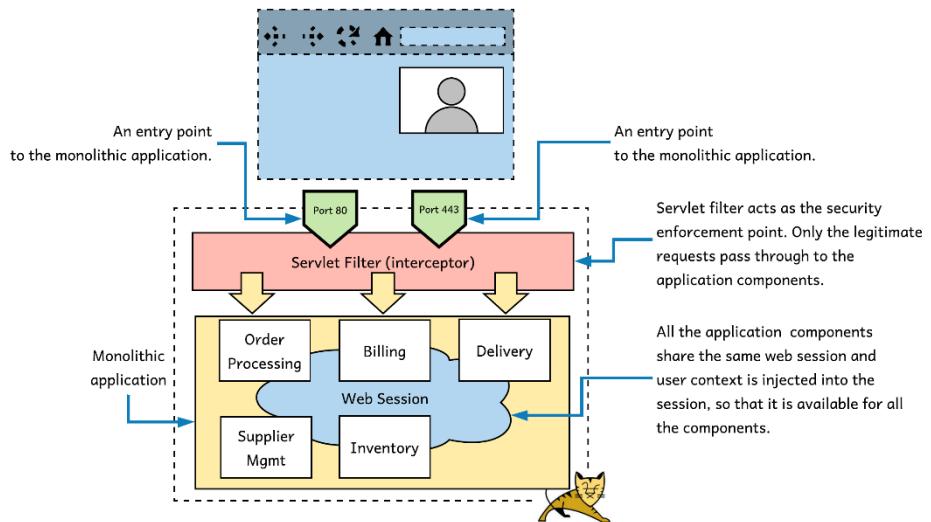


Figure 1.2 Multiple entry points (ports 80 and 443) are funneled to a single servlet filter. The filter acts as a centralized policy enforcement point.

Once a request is inside the application layer, you need not worry about security when one component talks to another. When the Order Processing component talks to the Inventory component, for example, you don't necessarily need to enforce any additional security checks (but of course you can if you need to enforce more granular access-control checks at component level). These calls are in-process calls and in most cases are hard for a third party to intercept. In most monolithic applications, security is enforced centrally, and individual components need not worry about carrying out additional checks unless there is a desperate requirement to do so. As a result, the security model of a monolithic application is much more straightforward than that of an application built around microservices architecture.

1.2 Challenges of securing microservices

Mostly due to the inherent nature of microservices architecture, security is challenging. In this section, we discuss the challenges of securing microservices without discussing in detail how to overcome them. In the rest of the book, we discuss multiple ways to address these challenges.

1.2.1 The broader the attack surface, the higher the risk of attack

In a monolithic application, communication among internal components happens within a single process—in a Java application, for example, within the same Java Virtual Machine (JVM). Under microservices architecture, those internal components are designed as separate, independent microservices, and those in-process calls among internal components become remote calls. Also, each microservice now independently accepts requests or has its own entry points (see figure 1.3). Instead of a couple of entry points, as in a monolithic application, now you have a large number of entry points. As the number of entry points to the system increases, the attack surface broadens too. This situation is one of the fundamental challenges in building a security design for microservices. Each entry point to each microservice must be protected with equal strength. The security of a system is no stronger than the strength of its weakest link.

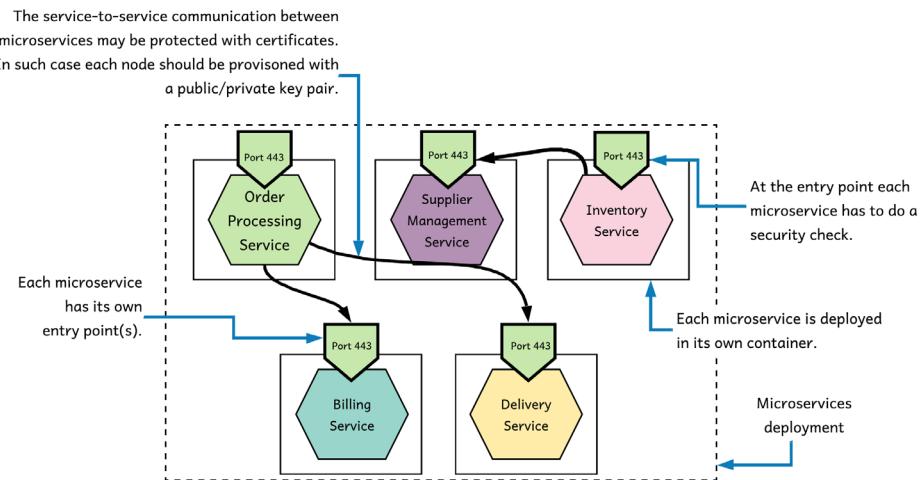


Figure 1.3 As opposed to a monolithic application with few entry points, a microservice-based application has many entry points, and they all must be secured.

1.2.2 Distributed security screening may result in poor performance

Unlike in a monolithic application, each microservice in a microservices deployment has to carry out independent security screening. From the viewpoint of a monolithic application, in which the security screening is done once and the request is dispatched to the corresponding component, having multiple security screenings at the entry point of each microservice seems to be redundant. Also, while validating requests at each microservice, you may need to connect to a remote security token service (STS). These repetitive, distributed security checks and remote connections could contribute heavily to latency and considerably degrade the performance of

the system. Some do work-around this by simply trusting the network and avoid security checks at each and every microservice. Over the time trust-the-network has become an anti-pattern and the industry is moving towards zero trust networking principles. With the zero trust networking principles you carry out security as much as closer to each resource in your network. Any microservices security design must take overall performance into consideration and must take precautions to address any drawbacks.

1.2.3 Deployment complexities make bootstrapping trust among microservices a nightmare

Security aside, how hard would it be to manage 10, 15, or hundreds of independent microservices instead of one monolithic application in a deployment? Not just in hundreds, but we have started seeing microservices deployments with thousands of services talking to each other. Capital One, one of the leading financial institutes in USA recently (July, 2019) announced that their microservices deployment consist of thousands of microservices on several thousands of containers, with thousands of AWS EC2 instances. Monzo, another financial institute based out of UK, recently mentioned that they have more than 1500 services running in their microservices deployment. Jack Kleeman, a backend engineer at Monzo, explains in this blog (<https://monzo.com/blog/we-built-network-isolation-for-1-500-services>), how they built network isolation for 1,500 services to make Monzo more secure. The bottom line is, the large-scale microservices deployments with thousands of services now have become a reality.

Managing a large-scale microservices deployments with thousands of services would be extremely challenging unless you know how to automate. If the microservices concept had popped up at a time when the concept of containers didn't exist, few people or organizations would have the guts to adopt microservices. Fortunately, things didn't happen that way, and that's why we believe that microservices and containers are a match made in heaven. If you're new to containers or don't know what Docker is, think about containers as being a way to make the software distribution and deployment hassle-free. Microservices and containers (Docker) were born at the right time to complement each other nicely. We talk about containers and Docker later in the book, under appendix A and chapter 10.

Would the deployment complexity of microservices architecture make security more challenging? We're not going to delve deep into the details here, but consider one simple example. Service-to-service communication happens among multiple microservices. Each of these communication channels must be protected. You have many options (which we discuss in detail later in the book under chapter 6 and chapter 7), but suppose that you use certificates. Now each microservice must be provisioned with a certificate (and the corresponding private key), which it will use to authenticate itself to another microservice during service-to-service interactions. The recipient microservice must know how to validate the certificate associated with the calling microservice. Therefore, you need a way to bootstrap trust between microservices. Also, you need to be able to revoke certificates (in case the corresponding private key gets compromised) and rotate certificates (change them periodically). These tasks are

cumbersome, and unless you find a way to automate them, they'll be tedious in a microservices deployment.

1.2.4 Requests which span across multiple microservices are harder to trace

Observability is a measure of what you can infer about the internal state of a system based on the external outputs that you observe. Logs, metrics, and traces are known as the three pillars of observability. A log can be any event you record that corresponds to a given service. A log, for example, can be an audit trail that says that the Order Processing microservice accessed the Inventory microservice to update the inventory on December 1, 2019, at 10:15.12 p.m. on behalf of the user Peter.

Aggregating a set of logs can produce metrics. In a way, metrics reflect the state of the system. In terms of security, the average invalid access requests per hour is a metric, for example. If the number is high, it probably indicates that the system is under attack or the first-level defense is weak. You can configure alerts based on metrics. If the number of invalid access attempts for a given microservice goes beyond a preset threshold value, the system can trigger an alert.

Tracing is also based on logs but provides a different perspective of the system. Tracing helps you track a request from the point where it enters the system to the point where it leaves the system. This process becomes challenging in a microservices deployment. Unlike in a monolithic application, a request to a microservice deployment may enter the system via one microservice and span multiple microservices before it leaves the system. Correlating requests among microservices is challenging, and you have to rely on distributed tracing systems like Jaeger and Zipkin. In chapter 5 we discuss how to use Prometheus and Grafana to monitor all the requests coming to a microservices deployment and in appendix F we discuss observability concepts in general with respect to microservices.

1.2.5 Immutability of containers challenges how you maintain service credentials and access-control policies

If a server is immutable, it doesn't change its state after it spins up, so it's called an *immutable server*. The most popular deployment pattern for microservices is container-based. (We use the terms *container* and *Docker* interchangeably in this book, and in this context, both terms have the same meaning.) Each microservice runs in its own container, and as a best practice, the container has to be an immutable server. In other words, after the container has spun up, it shouldn't change any of the files in its file system or maintain any runtime state within the container itself. The whole purpose of expecting servers to be immutable in a microservices deployment is to make deployment clean and simple. At any point, you can kill a running container and create a new one with the base configuration without worrying about runtime data. If the load on a microservice is getting high, for example, you need more server instances to scale horizontally. Because none of the running server instances maintains any runtime state, you can simply spin up a new container to share the load.

What impact does immutability have on security, and why do immutable servers make microservices security challenging? In microservices security architecture, a microservice itself becomes a security enforcement point.³ As a result, you need to maintain a list of whitelisted clients (probably other microservices) that can access the given microservice, and you need a set of access-control policies. These lists aren't static; both the whitelisted clients and access-control policies get updated. With an immutable server, you can't maintain such updates in the server's file system. You need a way to get all the updated policies from some sort of policy administration endpoint at server bootup and then update them dynamically in memory, following a push or pull model. In the push model, the policy administration endpoint pushes policy updates to the interested microservices (or security enforcement points). In the pull model, each microservice has to poll the policy administration endpoint periodically for policy updates.

Each microservice also has to maintain its own credentials, such as certificates. For better security, these credentials need to be rotated periodically. It's fine to keep them with the microservice itself (in the container file system), but you should have a way to inject them into the microservice at the time it boots up. Maybe this process can be part of the continuous delivery pipeline.

1.2.6 The distributed nature of microservices makes sharing user context harder

In a monolithic application all internal components share the same web session, and anything related to the requesting party (or user) is retrieved from it. In microservices architecture, you don't enjoy that luxury. Nothing is shared among microservices (or only a very limited set of resources), and the user context has to be passed explicitly from one microservice to another. The challenge is to build trust between two microservices so that the receiving microservice accepts the user context passed from the other one. You need a way to verify that the user context⁴ passed among microservices isn't deliberately modified. Using a JSON Web Token (JWT) is one popular way to share user context among microservices, and we explore this technique later in the book under chapter 7 and we discuss JWT in detail in appendix H. For now you can think of JWT as a JSON message, which helps carrying a set of user attributes from one microservice to another in a cryptographically safe manner.

³This isn't 100 percent precise, and we discuss why later in the book under chapter 12. In many cases, it's not the microservice itself that becomes the security enforcement point, but another proxy, which is deployed collocated with the microservice itself. Still, the argument we present here related to immutability is valid.

⁴User context carries information related to the user who invokes a microservice. This user can be a human user or a system, and the information related to the user can be a name, email address, or any other user attribute.

1.2.7 Polyglot architecture demands more security expertise on each development team

In a microservices deployment, services talk to one another over the network. They depend not on each service's implementation, but on the service interface. This situation permits each microservice to pick its own programming language and the technology stack for implementation. In a multi-team environment, in which each team develops its own set of microservices, each team has the flexibility to pick the most optimal technology stack for its requirements. This architecture, which promotes different components in a system to pick the technology stack what is best of itself, is known as the *polyglot architecture*.

The polyglot architecture makes security challenging. Because different teams use different technology stacks for development, each team has to have its own security experts. These experts should take responsibility for defining security best practices and guidelines, research security tools for each stack for static code analysis and dynamic testing, and integrate those tools into the build process. The responsibilities of a centralized, organization-wide security team are now distributed among different teams. In most cases, organizations use a hybrid approach, with a centralized security team and security-focused engineers on each team who build microservices.

1.3 Key security fundamentals

Adhering to fundamentals is important in any security design. There's no perfect or unbreakable security. How much you should worry about security isn't only a technical decision, but also an economic decision. There's no point in having a burglar-alarm system to secure an empty garage, for example. The level of security you need depends on the assets you intend to protect. The security design of an e-commerce application could be different from that of a financial application. In any case, adhering to security fundamentals is important. Even if you don't foresee some security threats, following the fundamentals helps you protect your system against such threats. In this section, we walk you through key security fundamentals and show you how they're related to microservices security.

1.3.1 Authentication protects your system against spoofing

Authentication is the process of identifying the requesting party to protect your system against spoofing. The requesting party can be a system (a microservice) or a system requesting access on behalf of a human user or another system (see figure 1.4). It's rather unlikely that a human user will access a microservice directly, though. Before doing a security design for a given system, you need to identify the audience. The authentication method you pick is based on the audience.

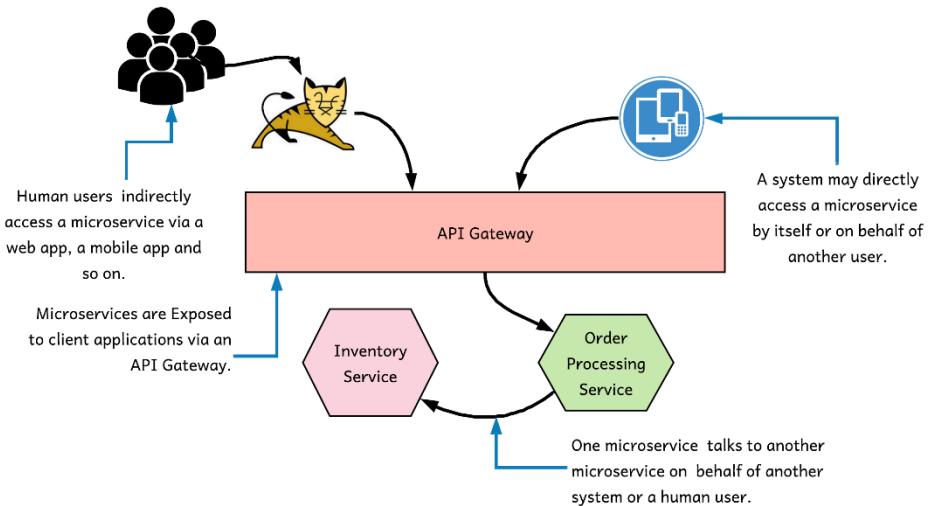


Figure 1.4 A system (a web/mobile application, etc) just by being itself or on behalf of a human user or another system, can access microservices via an API gateway.

If you’re worried about a system accessing a microservice on behalf of a human user, you need to think about how to authenticate the system as well as the human user. In practice, this can be a web application, which is accessing a microservice, on behalf of a human user who logs into the web application. In these kind of delegated use cases in which a system requests access on behalf of another system or a human user, OAuth 2.0 is the de facto standard. We discuss OAuth 2.0 in detail later in the book under appendix D.

To authenticate the human user to a system (for example a web application), you could request the username and password with some other factor for multifactor authentication (MFA). Whether MFA is required is mostly a business decision, based on how critical your business assets are or how sensitive the data you want to share with the users. The most popular form of MFA is the one-time passcode (OTP) sent over SMS. Even though it’s not the best method in terms of security, it’s the most usable form of MFA, mostly because a large portion of the world population has access to mobile phones (which don’t necessarily need to be smart phones). MFA helps to reduce account breaches by almost 99.99 percent⁵. Much stronger form of MFA includes, biometrics, certificates, FIDO (Fast Identity Online), and so on.

⁵ Basics and Black Magic: Defending against Current and Emerging Threats: <https://www.youtube.com/watch?v=Nmkgeg0wPRGE>

You have multiple ways to authenticate a system. The most popular options are certificates and JWTs. We discuss both these options in detail, with a set of examples, in chapter 6 and chapter 7.

1.3.2 Integrity protects your system from data tampering

When you transfer data from your client application to a microservice or from one microservice to another microservice, based on the strength of the communication channel you pick, an intruder could intercept the communication and change the data for his or her advantage. If the channel carries data corresponding to an order, for example, the intruder could change its shipping address to his or her own. Systems protected for integrity don't ignore this possibility; they introduce measures so that if a message is altered, the recipient can detect and discard the request.

The most common way to protect a message for integrity is to sign it. Any data in transit over a communication channel protected with Transport Layer Security (TLS), for example, is protected for integrity. If you use HTTPS for communication among microservices, that communication is in fact HTTP over TLS, and your messages are protected for integrity while in transit.

Along with the data in transit, the data at rest must be protected for integrity. Of all your business data, audit trails matter most for integrity checks. An intruder who gets access to your system would be happiest if she can modify your audit trails to wipe out any evidence. In a microservices deployment based on containers, audit logs aren't kept at each node that runs the microservice; they're published in some kind of a distributed tracing system like Jaeger or Zipkin. You need to make sure that the data maintained in those systems is protected for integrity. One way is to periodically calculate the message digests of audit trails, encrypt them, and store them securely. In a research paper⁶, Gopalan Sivathanu, Charles P. Wright, and Erez Zadok of Stony Brook University highlight the causes of integrity violations in storage and present a survey of available integrity assurance techniques. The paper explains several interesting applications of storage integrity checking apart from security and discusses implementation issues associated with those techniques.

1.3.3 Nonrepudiation: Do it once, and you own it forever

Nonrepudiation is an important aspect of information security that prevents you from denying anything you've done or committed. Consider a real-world example. When you lease an apartment, you agree to some terms and conditions with the leasing company. If you leave the apartment before the end of the lease, you're obliged to pay the rent for the remaining period or find some other tenant to sublease the apartment. All the terms are in the leasing agreement, which you accept by signing it. After you sign it, you can't dispute the terms and conditions to

⁶ Ensuring Data Integrity in Storage: Techniques and Applications <https://www.fsl.cs.sunysb.edu/docs/integrity-storage05/integrity.html>

which you agreed. That's nonrepudiation in the real world. It creates a legal obligation. Even in the digital world, a signature helps you achieve nonrepudiation; in this case, you use a digital signature.

In an e-commerce application, for example, after a customer places an order, the Order Processing microservice has to talk to the Inventory microservice to update inventory. If this transaction is protected for nonrepudiation, the Order Processing microservice can't later deny that it updated inventory. If the Order Processing microservice signs a transaction with its private key, it can't deny later that the transaction was initiated from that microservice. With a digital signature, only the owner of the corresponding private key can generate the same signature; so make sure that you never lose the key!

Validating the signature alone doesn't help you achieve nonrepudiation, however. You also need to make sure that you record transactions along with the timestamp and the signature – and maintain those records for a considerable amount of time. In case the initiator disputes a transaction later, you'll have it in your records.

1.3.4 Confidentiality protects your systems from unintended information disclosure

When you send some order data from a client application to the Order Processing microservice, you expect that no party can view the data other than the Order Processing microservice itself. But based on the strength of the communication channel you pick, an intruder can intercept the communication and get hold of the data. Along with the data in transit, the data at rest needs to be protected for confidentiality (see figure 1.5). An intruder who gets access to your data storage or backups has direct access to all your business-critical data unless you've protected it for confidentiality.

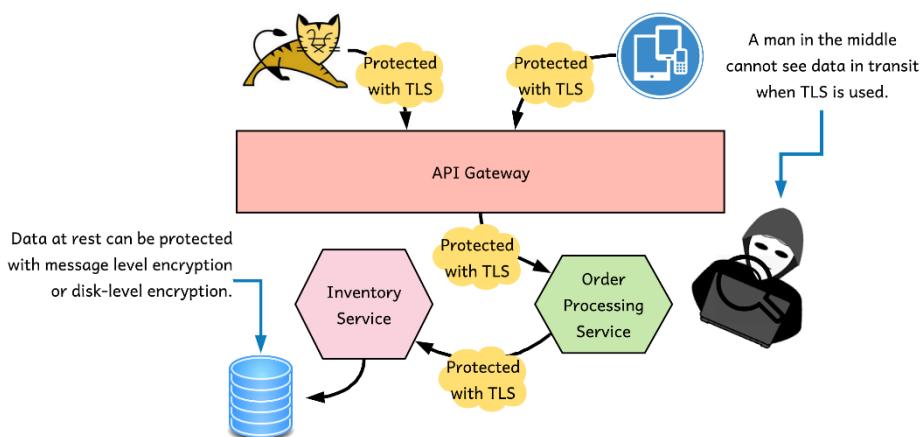


Figure 1.5 To protect a system for confidentiality, both the data in transit and at rest must be protected. The data in transit can be protected with TLS while data at rest by encryption.

DATA IN TRANSIT

Encryption helps you achieve confidentiality. A cryptographic operation makes sure that the encrypted data is visible only to the intended recipient. TLS is the most popular way of protecting data for confidentiality in transit. If one microservice talks to another over HTTPS, you're using TLS underneath, and only the recipient microservice will be able to view the data in cleartext. Then again, the protection provided by TLS is point-to-point. At the point where the TLS connection terminates, the security ends. If your client application connects to a microservice over a proxy server, your first TLS connection terminates at the proxy server, and a new TLS connection is established between the proxy server and the microservice. The risk is that anyone who has access to the proxy server can log the messages in cleartext as soon as the data leaves the first connection.

Most proxy servers support two modes of operations with respect to TLS: TLS bridging and TLS tunneling. TLS bridging terminates the first TLS connection with the proxy server, and creates a new TLS connection between the proxy server and the next destination of the message. If your proxy server uses TLS bridging, don't trust it and possibly put your data at risk, even though you use TLS (or HTTPS). If you use TLS bridging, the messages are in cleartext while transiting through the proxy server. TLS tunneling creates a tunnel between your client application and the microservices, and no one in the middle will be able to see what's going through, not even the proxy server. If you are interested in reading more about TLS, we recommend you having a look at the book, *SSL and TLS: Designing and Building Secure Systems* by Eric Rescorla (Addison-Wesley Professional, 2000)

NOTE Encryption has two main flavors: Public-key encryption and symmetric-key encryption. With public-key encryption, the data is encrypted using the recipient's public key, and only the party who owns the corresponding private key can decrypt the message and see what's in it. With symmetric key encryption, the data is encrypted with a key known to both the sender and the recipient. TLS uses both the flavors. Symmetric-key encryption is used to encrypt the data, while public-key encryption is used to encrypt the key used in symmetric-key encryption.

DATA AT REST

Encryption should also apply to data at rest to protect it from intruders who get direct access to the system. This data can be credentials for other systems stored in the file system or some business-critical data stored in a database. Most database management systems provide features for automatic encryption, and disk-level encryption features are available at operating-system level. Application-level encryption is another option, in which the application itself encrypts the data before passing it over to the file system or to a database. Of all these options, the one that best fits your application depends on the criticality of your business operations.

Also keep in mind that encryption is a resource-intensive operation⁷ that would have considerable impact on your application's performance unless you find the most optimal solution.

1.3.5 Availability: Keep the system running, no matter what

The whole point of building any kind of a system is to make it available to its users. Every minute (or even second) that the system is down, your business loses money. Amazon was down for 20 minutes in March 2016, and the estimated revenue loss was \$3.75 million. In January 2017, more than 170 Delta Airlines flights were canceled due to a system outage, which resulted in an estimated loss of \$8.5 million. It's not only the security design of a system that you need to worry about to keep a system up and running, but also the overall architecture. A bug in the core functionality of an application can take the entire system down. To some extent, these kinds of situations are addressed in the core design principles of microservices architecture. Unlike in monolithic applications, in a microservices deployment, the entire system won't go down if a bug is found in one component or microservice. Only that microservice will go down; the rest should be able to function.

Of all the factors that can take a system down, security has a key role to play in making a system constantly available to its legitimate stakeholders. In a microservices deployment, with a large number of entry points (which may be exposed to the Internet), an attacker can execute a denial of service (DoS) or a distributed denial of service (DDoS) attack and take the system down. Defenses against such attacks can be built on different levels. On the application level, the best thing you could do is reject a message (or a request) as soon as you find that it's not legitimate. Having layered security architecture helps you reject an attacker at the outermost layer. As shown in figure 1.6, any request to a microservice first has to come through the API gateway. The API gateway centrally enforces security for all the requests entering the microservices deployment, including authentication, authorization, throttling, and message content validation for known security threats. We get into the details on each topic later in the book under chapter 3, chapter 4 and chapter 5.

⁷ Performance Evaluation of Encryption Techniques for Confidentiality of Very Large Databases: <http://www.ijcte.org/papers/410-G1188.pdf>

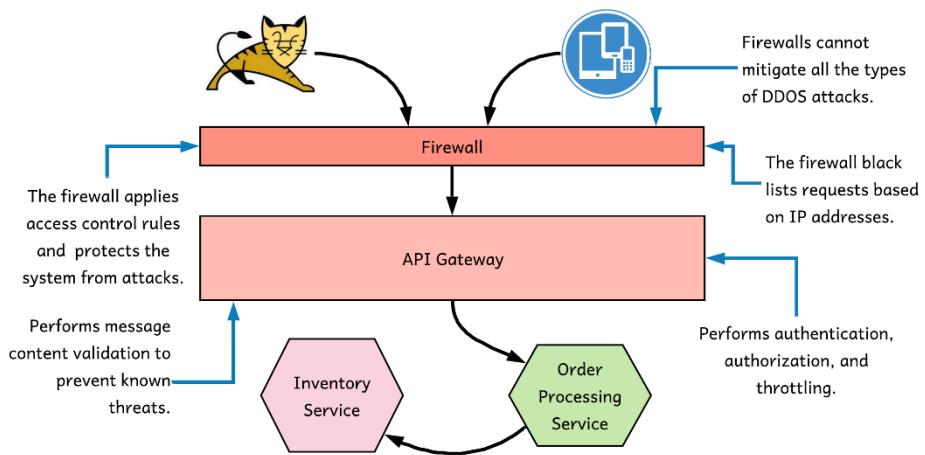


Figure 1.6 Multiple security enforcement points at multiple layers help improving the level of security of a microservices deployment.

The network perimeter level is where you should have the best defense against DoS/DDoS attacks. A firewall is one option; it runs at the edge of the network and can be used to keep malicious users away. But firewalls can't protect you completely from a DDoS attack. Specialized vendors provide DDoS prevention solutions for use outside a corporate firewalls. You need to worry about those solutions only if you expose your system to the Internet. Also, all the DDoS protection measures you take at the edge aren't specific to microservices. Any endpoint that's exposed to the Internet must be protected from DoS/DDoS attacks.

1.3.6 Authorization: Nothing more than you're supposed to do

Authentication helps you learn about the user or the requesting party. Authorization deals with what actions an authenticated user can perform on the system. In an e-commerce application, for example, any customer who logs into the system can place an order, but only the inventory managers can update the inventory. In a typical microservices deployment, authorization can happen at the edge (the entry point to the microservices deployment, which could be intercepted by a gateway) and at each service level. Later in this chapter, under section 1.4.3 we discuss how authorization policies are enforced at the edge and what options you have to enforce authorization policies in service-to-service communication at the service level.

1.4 Edge security

In a typical microservices deployment, microservices are not exposed directly to client applications. In most cases, microservices are behind a set of APIs that is exposed to the outside

world via an API gateway. The API gateway is the entry point to the microservices deployment, which screens all incoming messages for security. Figure 1.7 depicts a microservices deployment that resembles Netflix's, in which all the microservices are fronted by the Zuul⁸ API gateway. Zuul provides dynamic routing, monitoring, resiliency, security, and more. It acts as the front door to Netflix's server infrastructure, handling traffic from Netflix users around the world. In figure 1.7, Zuul is used to expose the Order Processing microservice via an API. Other microservices in the deployment, Inventory and Delivery, don't need to be exposed from the API gateway because they don't need to be invoked by external applications. In a typical microservices deployment there can have a set of microservices that external applications can access and another set of microservices that external applications don't need to access; only the first set of microservices is exposed to the outside world via an API gateway.

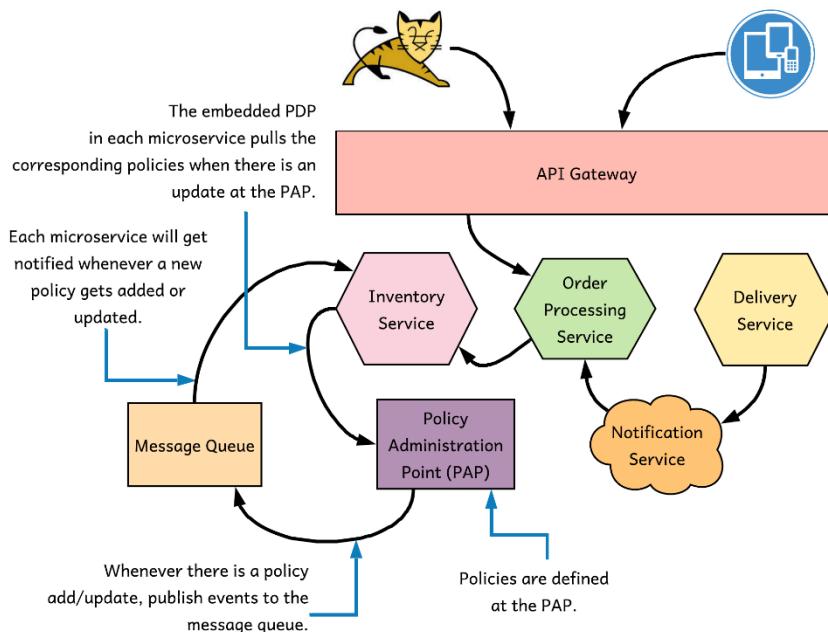


Figure 1.7 A typical microservices deployment with an API gateway, where the API gateway is the entry point, which screens all incoming messages for security

⁸ Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more: <https://github.com/Netflix/zuul>

1.4.1 The role of an API gateway in a microservices deployment

Over time, APIs have become the public face of many companies. We're not exaggerating by saying that a company without an API is like a computer without the Internet. If you're a developer, you surely know how the life would look like with no Internet! APIs have also become many companies' main revenue-generation channel. At Expedia, for example, 90 percent of revenue comes through APIs; at Salesforce, APIs account for 50 percent of revenue; and at eBay, APIs account for 60 percent of revenue.⁹ Netflix is another company that has heavily invested in APIs. Netflix accounts for one third of all Internet traffic in North America, all of which comes through Netflix APIs.¹⁰

APIs and microservices go hand in hand. Any microservice that needs to be accessed by a client application is exposed as an API via an API gateway. The key role of the API gateway in a microservices deployment is to expose a selected set of microservices to the outside world as APIs and build quality of service (QoS) features. These QoS features are security, throttling, and analytics. Exposing a microservice to the outside world, however, doesn't necessarily mean making it public-facing or exposed to the Internet. You could expose it only outside your department, allowing users and systems from other departments within the same organizational boundary to talk to the upstream microservices via an API gateway. Later in the book under chapter 3, we discuss in detail the role that an API gateway plays in a microservices deployment.

1.4.2 Authentication at the edge

As for microservices, for APIs the audience is a system that acts on behalf of itself or on behalf of a human user or another system (see figure 1.8). It's unlikely (but not impossible) for human users to interact directly with APIs. In most cases, an API gateway deals with systems. In the following sections, we discuss options for authenticating a system at the API gateway.

⁹ The Strategic Value of APIs: <https://hbr.org/2015/01/the-strategic-value-of-apis>

¹⁰ Half of All Internet Traffic Goes to Netflix and YouTube: <http://testinternetspeed.org/blog/half-of-all-internet-traffic-goes-to-netflix-and-youtube/>

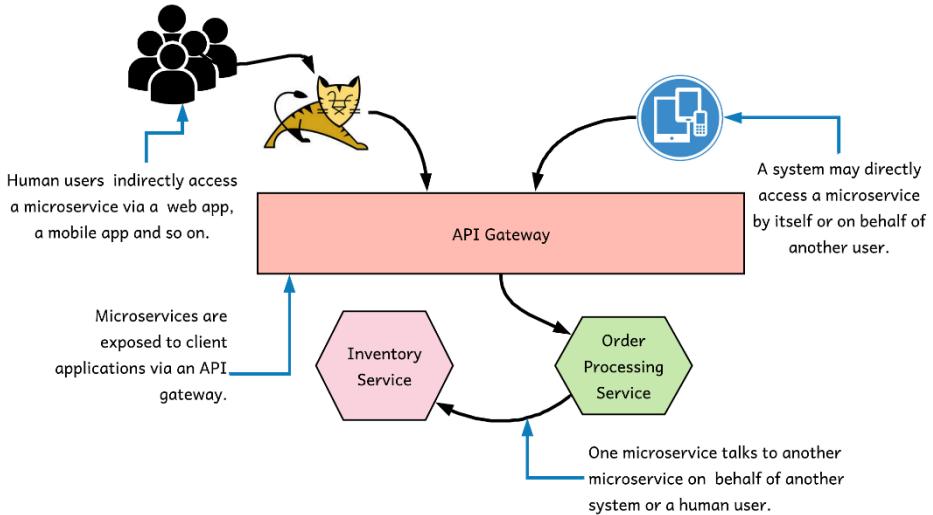


Figure 1.8 Authentication at the edge is enforced by the API gateway. Only the authenticated requests are dispatched to the upstream microservices.

CERTIFICATE-BASED AUTHENTICATION

Certificate-based authentication protects an API at the edge with mutual Transport Layer Security (mTLS). In the Netflix microservices deployment, access to the APIs is protected with certificates. Only a client provisioned with a valid certificate can access Netflix APIs. The role of the API gateway here is to make sure that only clients carrying a trusted certificate can access the APIs and that only those requests are routed to the upstream microservices. Later in the book, we discuss how to secure APIs at the API gateway with mTLS, under chapter 3.

OAuth 2.0-BASED ACCESS DELEGATION

Anyone can create an application to consume Twitter and Facebook APIs. These applications can be web or mobile (refer to figure 1.8). An application can access an API as itself or on behalf of a human user. OAuth 2.0, which is an authorization framework for delegated access control, is the recommended approach for protecting APIs when one system wants to access an API on behalf of another system or a user.

We explain OAuth 2.0 in detail later in the book under chapter 3 and appendix D; don't worry if you don't know what it is. Even if you don't know what OAuth 2.0 is, you use it if you use Facebook to log in to third-party web applications, because Facebook uses OAuth 2.0 to protect its APIs.

Cambridge Analytica/Facebook scandal

The Cambridge Analytica/Facebook privacy scandal happened in early 2018, when Facebook discovered that the personal data of more than 87 million people collected by a third-party application called This Is Your Digital Life, created by a researcher called Alexander Kogan, was sold to a company called Cambridge Analytica. The third-party application acted as a system, accessing the Facebook API secured with OAuth 2.0, on behalf of legitimate Facebook users to collect their personal data. In other words, these Facebook users directly or indirectly delegated access to their personal data to this third-party application.

The role of the API gateway is to validate the OAuth 2.0 security tokens that come with each API request. The OAuth 2.0 security token represents both the third-party application and the user who delegated access to the third-party application to access an API on his or her behalf.

Those who know about OAuth 2.0 probably are raising their eyebrows at seeing it mentioned in a discussion of authentication. We agree that it's not an authentication protocol at the client application end, but at the resource server end, which is the API gateway. We discuss this topic later in the book under appendix D.

1.4.3 Authorization at the edge

In addition to figuring out who the requesting party is during the authentication process, the API gateway could enforce corporatewide access-control policies, which are probably coarse grained. More fine-grained access control policies are enforced at the service level by the microservice itself. Later in the chapter, under section 1.5.2 we discuss service-level authorization in detail.

1.4.4 Passing client/end-user context to upstream microservices

The API gateway terminates all the client connections at the edge, and if everything looks good, it dispatches the requests to the corresponding upstream microservices. But you need a way to protect the communication channels between the gateway and the corresponding microservice, as well as a way to pass the initial client/user context. User context carries basic information about the end user, and client context carries information about the client application. This information probably could be used by upstream microservices to do service-level access control.

As you may have rightly guessed, communication between the API gateway and the microservices is system-to-system, so you probably can use mTLS authentication to secure the channel. But how do you pass the user context to the upstream microservices? You have a couple of options: pass the user context in an HTTP header, or create a JWT with the user data. The first option is straightforward but raises some trust concerns when the first microservice passes the same user context in an HTTP header to another microservice. The second microservice doesn't have any guarantee that the user context isn't altered. But with JWT, you have an assurance that a man in the middle can't change its content and go undetected, because the issuer of the token signs it.

We explain JWT in detail under appendix H; for now, think of it as a signed payload that carries some data (in this case, the user context) in a cryptographically safe manner. The gateway or a security token service connected to the gateway can create a JWT that includes the user context (and the client context) and passes it to the upstream microservices. The recipient microservices can validate the JWT by verifying the signature with the public key of the security token service that issued the JWT.

1.5 Securing service-to-service communication

The frequency of service-to-service communication is higher in a microservices deployment. Communication can occur between two microservices within the same trust domain or between two trust domains. A trust domain represents the ownership. Microservices developed, deployed, and managed together probably fall under one trust domain, or the trust boundaries can be defined at the organizational level by taking many other factors into account. The security model that you develop to protect service-to-service communication should consider the communication channels that cross trust boundaries, as well as how the actual communication takes place between microservices: synchronously or asynchronously. In most cases, synchronous communication happens over HTTP. Asynchronous communication can happen over any kind of messaging system, such as RabbitMQ, Kafka, ActiveMQ, or even Amazon SQS. In chapter 6, chapter 7 and chapter 8 we discuss different security models to secure synchronous communication happens among microservices and chapter 9 talks about securing event-driven microservices.

1.5.1 Service-to-service authentication

You have three common ways to secure communication among services in a microservices deployment: trust the network, mutual transport layer security (mTLS), and JSON Web Tokens (JWTs).

The *trust-the-network* approach is an old-school model in which no security is enforced in service-to-service communication; rather, the model relies on network-level security (see figure 1.9). Network-level security must guarantee that no attacker can intercept communication among microservices. Also, each microservice is a trusted system. Whatever it claims about itself and the end user is trusted by other microservices. You should make this deployment choice based on the level of security you expect and the trust you keep on every component in the network.

Another school of thought, known as *zero-trust network*, opposes the trust-the-network approach. The zero-trust network approach assumes that the network is always hostile and untrusted, and it never takes anything for granted. Each request must be authenticated and authorized at each node before being accepted for further processing. If you are interested in reading more about zero-trust networks, we recommend you having a look at the book, *Zero Trust Networks: Building Secure Systems in Untrusted Networks* by Evan Gilman and Doug Barth (O'Reilly Media, 2017).

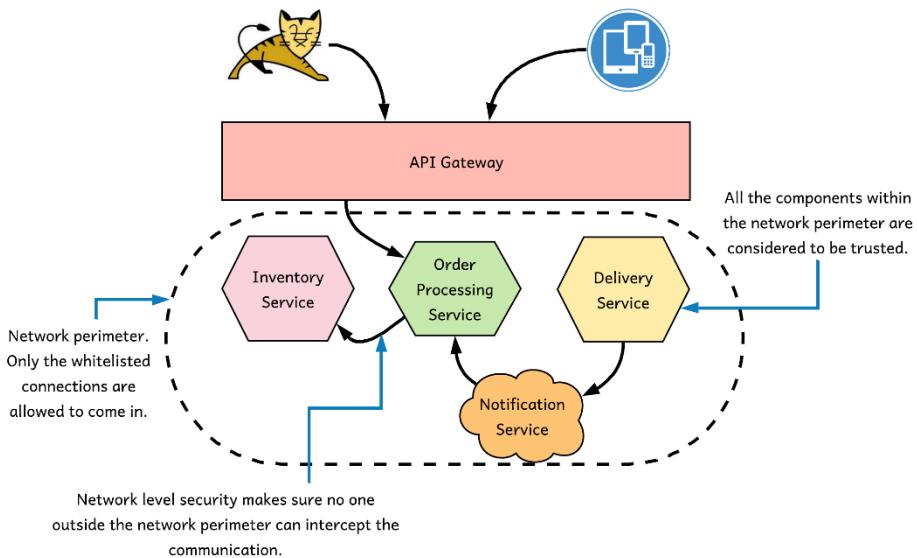


Figure 1.9 The trusted network makes sure that communication among microservices is secured. No one or a system outside the trusted network can see the traffic flows among microservices in the trusted network.

Mutual TLS (mTLS) is another popular way to secure service-to-service communications in a microservices deployment (see figure 1.10). In fact, this method is the most common form of authentication used today. Each microservice in the deployment has to carry a public/private key pair and uses that key pair to authenticate to the recipient microservices via mTLS. TLS provides confidentiality and integrity for the data in transit, and helps the client identify the service. The client microservice knows which microservice it's going to talk with. But with TLS (one-way), the recipient microservice can't verify the identity of the client microservice. That's where mTLS comes in. mTLS lets each microservice in communication identify the others. Challenges in mTLS include trust bootstrap and provisioning keys/certificates to workloads/microservices, key revocation, key rotation, and monitoring key use. We discuss those challenges and possible solutions in detail later in the book under chapter 6.

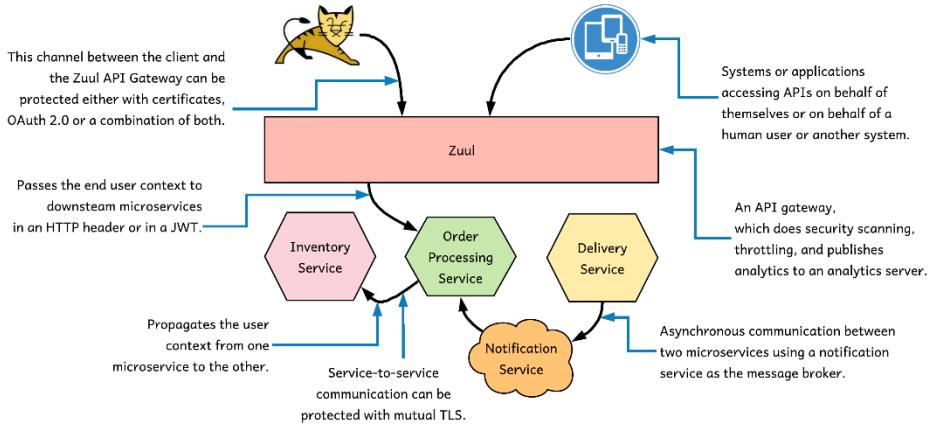


Figure 1.10 Communication among microservices is secured with mTLS. All the microservices communicate with each other trust the certificate authority (CA) in the deployment.

JSON Web Token (JWT) is the third approach for securing service-to-service communication in a microservices deployment (see figure 1.11). Unlike mTLS, JWT works at the application layer, not at the transport layer. JWT is a container that can carry a set of claims from one place to another. These claims can be anything, such as end-user attributes (email address, phone number), end-user entitlements (what he or she can do), or anything the calling microservice wants to pass to the recipient microservice. The JWT includes these claims and is signed by the issuer of the JWT. The issuer can be an external security token service (STS) or the calling microservice itself. The latter example is a self-issued JWT. As in mTLS, if we use self-issued JWT-based authentication, each microservice must have its own key pair, and the corresponding private key is used to sign the JWT. In most cases, JWT-based authentication works over TLS; JWT provides authentication, and TLS provides confidentiality and integrity of data in transit.

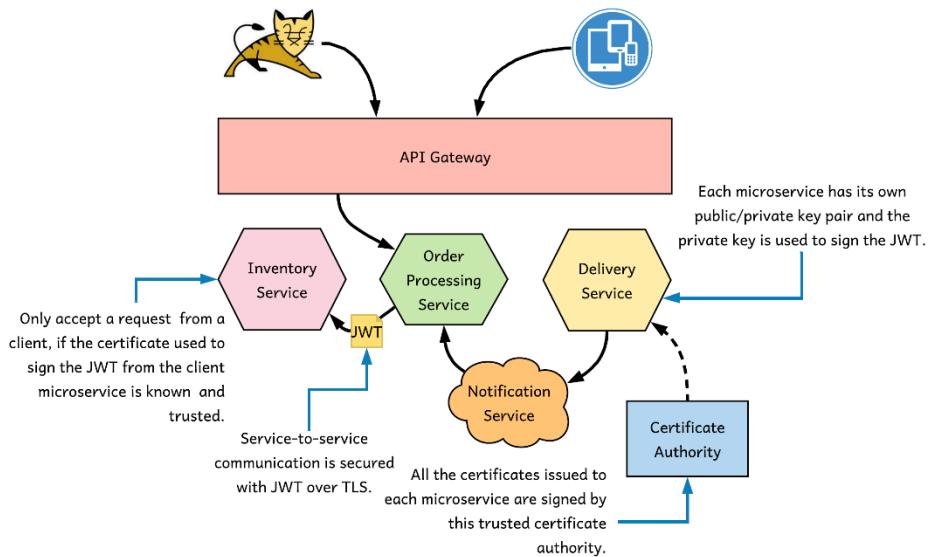


Figure 1.11 Communication among microservices is secured with JWT. Each microservice uses a certificate issued to it by the certificate authority to sign JWTs.

1.5.2 Service-level authorization

In a typical microservices deployment, authorization can happen at the edge (with the API gateway), at the service, or in both places. Authorization at service level gives each service the more control to enforce access control policies in the way it wants. Two approaches are used to enforce authorization at service level: the centralized policy decision point (PDP) model and the embedded PDP model. In the centralized PDP model, all the access-control policies are defined, stored, and evaluated centrally (see figure 1.12). Each time the service wants to validate a request, it has to talk to an endpoint exposed by the centralized PDP. This method creates a lot of dependency on the PDP and also badly affects latency due to the cost of calling the remote PDP endpoint. In some cases, the effect on latency can be prevented by caching policy decisions at service level, but other than cache expiration time, there's no way to communicate policy update events to the service. In practice, policy updates happen less frequently, and cache expiration may work in most cases.

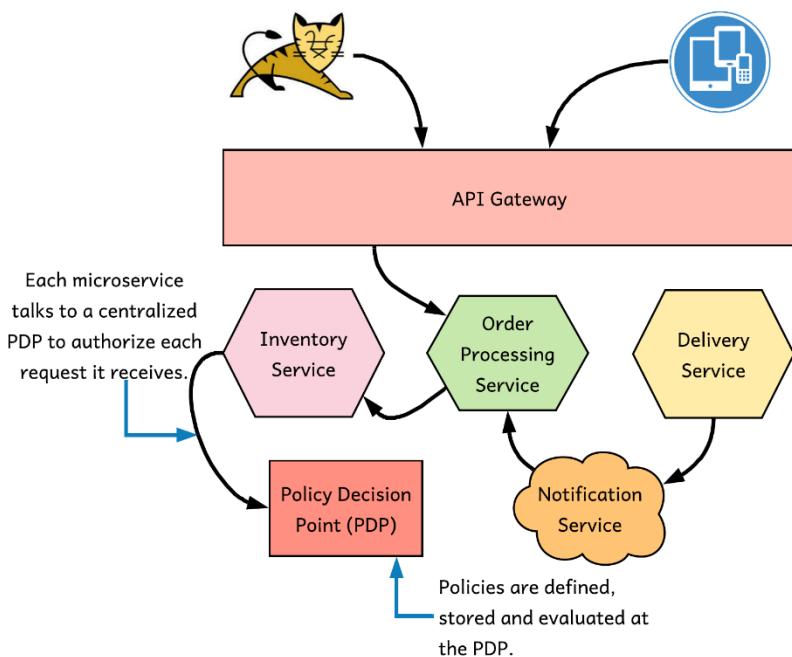


Figure 1.12 Each microservice is connected to a centralized PDP to authorize requests. All the access-control policies are defined, stored, and evaluated centrally

With embedded PDPs, policies are defined centrally but are stored and evaluated at service level. The challenge with embedded PDPs is how to get policy updates from the centralized policy administration point (PAP). There are two common methods. One approach is to poll the PAP continuously after a set period and then pull new and updated policies from PAP. The other approach is based on a push mechanism. Whenever a new policy or policy update is available, the PAP publishes an event to a topic (see figure 1.13). Each microservice acts as an event consumer and registers for the events it's interested in. Whenever a microservice receives an event for a registered topic, it pulls the corresponding policy from the PAP and updates the embedded PDP. Some people believe that both these approaches are overkill, however; they load policies to the embedded PDP only when the server starts up from a shared location. Whenever a new policy or a policy update is available, each service has to restart.

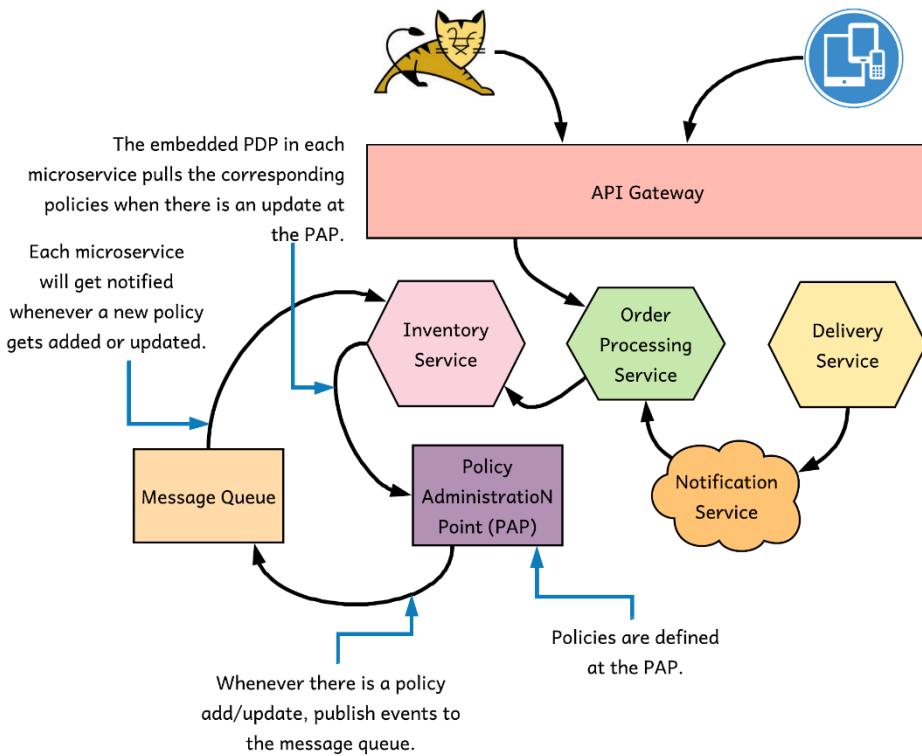


Figure 1.13 Each microservice embeds a PDP. The embedded PDPs pull the policies from the policy administration point upon receiving a notification.

1.5.3 Propagating user context between microservices

When one microservice invokes another microservice, it needs to carry both the end-user identity and the identity of the microservice itself. When one microservice authenticates to another microservice with mTLS or JWT, the identity of the calling microservice can be inferred from the embedded credentials. There are three common ways to pass the end-user context from one microservice to another microservice:

- *Send the user context as an HTTP header.* This technique helps the recipient microservice identify the user but still has to trust the calling microservice. If the calling microservice wants to fool the recipient microservice, it can do so easily by setting any name it wants as the HTTP header.
- *Use a JWT.* This JWT carries the user context from the calling microservice to the recipient microservice and is also passed in the HTTP request as a header. This approach has no

extra value in terms of security over the first approach if the JWT that carries the user context is self-issued. A self-issued JWT is signed by the calling service itself, so it can fool the recipient microservice by adding any name it wants to add.

- Use a JWT issued by an external STS that is trusted by all the microservices in the deployment. The user context included in this JWT can't be altered, as alteration would invalidate the signature of the JWT. This is the most secure approach. When you have the JWT from an external STS, the calling microservice can embed it in the new JWT it creates to make a nested JWT (if JWT-based authentication is used between microservices) or pass the original JWT as it is as an HTTP header (if mTLS is being used between microservices).

1.5.4 Crossing trust boundaries

In a typical microservices deployment, you find multiple trust domains. We can define these trust domains by the teams having control and governance over the microservices or organizational boundaries. The purchasing department, for example, might manage all its microservices and create its own trust domain. In terms of security, when one microservice talks to another microservice, and if both of microservices are in the same trust domain, each microservice may trust one STS in the same domain or a certificate authority in the same domain. Based on this trust, the recipient microservice can validate a security token sent to it by a calling microservice. Typically, in a single trust domain, all the microservices trust one STS and accept only security tokens issued by that STS.

When one microservice wants to talk to another microservice in a different trust domain, it can take one of two primary approaches. In the first approach (see figure 1.14), the calling microservice (Order Processing) in the `foo` trust domain wants to talk to the recipient microservice (Delivery) of the `bar` trust domain. First, it has to obtain a security token that is trusted by all the microservices in the `bar` trust domain. In other words, it needs to obtain a security token from the STS of the recipient trust domain.

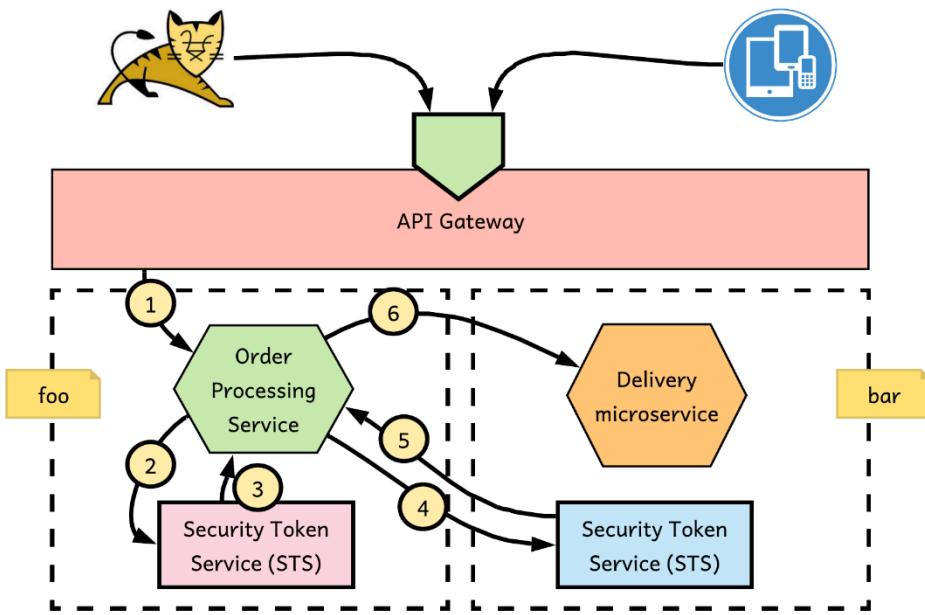


Figure 1.14 Cross-domain security between two trust domains behind a single trusted API gateway (and a security token service). Each trust domain has its own security token service.

Here's the numbered flow shown in figure 1.14:

- *Step 1*—The API gateway routes the request from the client application to the Order Processing microservice in the `foo` trust domain, along with a JWT signed by the gateway (or by an STS attached to it). Because all the microservices in the `foo` trust domain trust the top-level STS, the Order Processing microservice accepts the token as valid. The JWT has an attribute called `aud` that defines the target system of the JWT. In this case, the value of `aud` is set to the Order Processing microservice of the `foo` trust domain. Ideally, if the Order Processing microservice receives a JWT with a different `aud` value, it must reject that JWT, even if its signature is valid. We discuss JWT in detail in appendix H.
- *Step 2*—The Order Processing microservice passes the original JWT that it got from the gateway (or STS at the top level) to the STS at the `foo` trust domain. Once again, the `foo` STS has to validate the `aud` value in the JWT it gets. If it cannot identify the audience of the token, must reject it.
- *Step 3*—The `foo` STS returns a new JWT, which is signed by it and has an `aud` value targeting the STS in the `bar` trust domain.
- *Steps 4 and 5*—The Order Processing microservice accesses the STS of the `bar` trust domain and exchanges the JWT from step 3 to a new JWT signed by the STS of the `bar`

trust domain, with an `aud` value targeting the Delivery microservice.

- *Step 6*—The Order Processing microservice accesses the Delivery microservice with the JWT obtained from steps 4 and 5. Because the STS of the `bar` domain signs this JWT and has a matching `aud` value, the Delivery microservice will accept the token.

In the second approach, the Order Processing microservice from the `foo` trust domain doesn't talk directly to the Delivery microservice of the `bar` trust domain. Each trust domain has its own API gateway, and communication among microservices happens via the gateways (see figure 1.15).

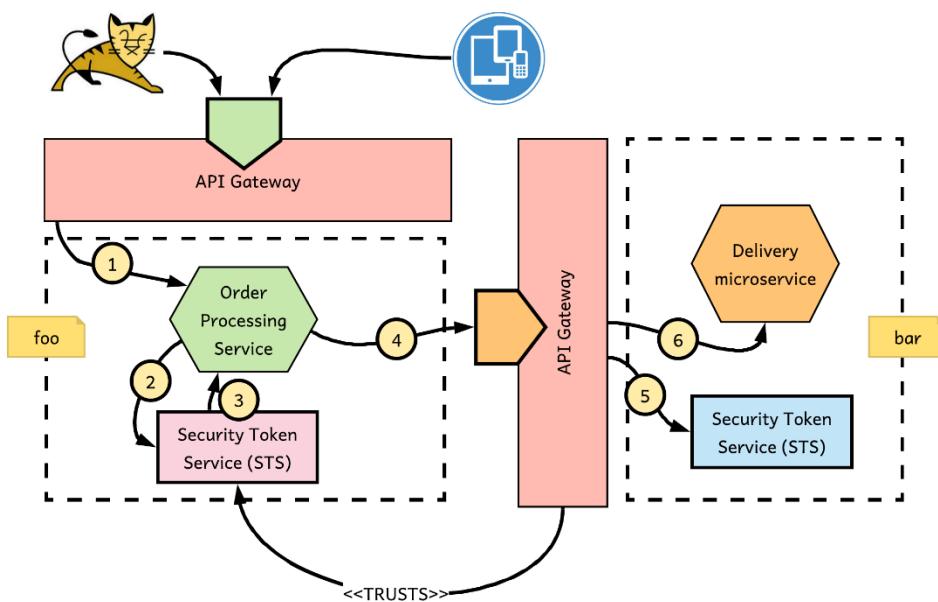


Figure 1.15 Cross-domain security between two trust domains behind two API gateways (and STSes)

Here's the numbered flow shown in figure 1.15:

- *Step 1*—The API gateway of the `foo` trust domain routes the request from the client application to the Order Processing microservice, along with a JWT signed by the gateway (or the `foo` STS). Because all the microservices in the `foo` trust domain trust the `foo` STS, the Order Processing microservice accepts the token as valid.
- *Step 2*—The Order Processing microservice passes the original JWT that it got from the gateway (or the `foo` STS) to its own STS (which is also the `foo` STS).
- *Step 3*—The `foo` STS returns a new JWT, which is signed by it and has an `aud` value targeting the API gateway of the `bar` trust domain.

- *Step 4*—The Order Processing microservice accesses the Delivery microservice of the `bar` domain with the JWT obtained from step 3. Because the API gateway of the `bar` domain trusts the `foo` domain STS, it accepts the JWT as valid. The JWT is signed by the `foo` STS and has an `aud` value to match the `bar` API gateway.
- *Step 5*—The `bar` API gateway talks to the `bar` STS to create its own JWT (signed by the `bar` STS) with an `aud` value to match the Delivery microservice.
- *Step 6*—The `bar` API gateway forwards the request to the Delivery microservice along with the new JWT issued by the `bar` STS. Because the Delivery microservice trusts its own STS, the token is accepted as valid.

1.6 Summary

- Securing microservices is quite challenging with respect to securing a monolithic application, mostly due to the inherent nature of the microservices architecture.
- There are four main areas in a microservices security design. It starts by defining a process to streamline the development process and engage security-scanning tools to the build system, so that we can discover the code level vulnerabilities at very early stage in the development cycle.
- We need to worry about edge security and securing communications between microservices.
- The edge security is about authenticating and authorizing requests coming into the microservices deployment, at the edge, probably with an API gateway.
- Securing the communication between microservices is the most challenging part. We discussed multiple techniques in this chapter, and what you have to pick will depend on many factors, such as the level of security, the type of communication (synchronous or asynchronous), trust boundaries and many more.

2

First steps in securing microservices

This chapter covers

- Developing a microservice in Spring Boot/Java, running and testing it with curl
- Securing a microservice at the edge with OAuth 2.0
- Enforcing authorization at the service-level with OAuth 2.0 scopes

You build applications as a collection of smaller/modular services or components, when you adhere to architectural principles of microservices. A system by itself or a system on behalf of a human user or another system can invoke a microservice. In all three cases we need to properly authenticate and authorize all the requests that reach the microservice. A microservice may also consume one or more other microservices in order to cater a request. In such cases it is also necessary to propagate user context (from downstream services or client applications) to upstream microservices. In this chapter, we explain how the security validation of the incoming requests happens and later in the book in chapter 3, we discuss how to propagate the user context to upstream microservices. The focus of this chapter is to get you started with a quite straightforward deployment. The design of the samples presented in this chapter is far from a production deployment. As we proceed in the book, we explain how to fill the gaps and in step by step to build a production grade microservices security design.

2.1 Your first microservice

In this section of the chapter we discuss how we can write, compile and run our first microservice using Spring Boot. You will learn some basics about the Spring Boot framework and how you can use it to build microservices. Throughout this book we use a retail store

application as an example, which we build with a set of microservices. In this section we build our first microservice, which accepts requests to create and manage orders, using Spring Boot (<https://spring.io/projects/spring-boot>). Spring Boot is a framework based on the Spring platform that allows you to convert functions written in Java programming language to network-accessible functions known as services/APIs by decorating your code with a special set of annotations. If you're not familiar with Java, still there's nothing to worry about, because we don't expect you to write code yourself. All the code samples you see in this book are available on GitHub (<https://github.com/microservices-security-in-action/samples>). As long as you are or have been a software developer, you'll find it easy to understand the code. The figure 2.1 shows a set of microservices, which are part of the retail store application we are building, with a set of consumer applications. The consumer applications in fact, are the consumers of the microservices we build.

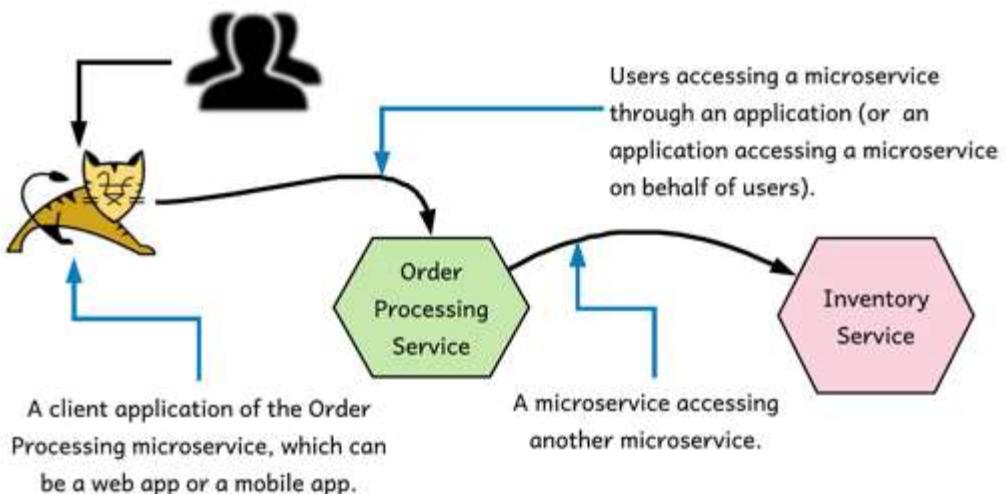


Figure 2.1 A typical microservices deployment, where consumer applications (a web app or a mobile app) accessing microservices on behalf their end users, while microservices communicate with each other.

2.1.1 Downloading and installing the required software

To build and run samples we use in this chapter and mostly in the rest of the book, you need to have a development environment setup with Java Development Kit (JDK), Apache Maven, curl command-line tool and Git command-line client.

INSTALLING THE JDK

JDK is required to compile the source code in the samples. You can download the latest JDK from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. We use Java version 11 to test all the samples.

INSTALLING APACHE MAVEN

Maven is a project management and comprehension tool that makes it easy to declare third-party (external) dependencies of your Java project required in the compile/build phase. It has various plugins such as the compiler plugin, which compiles your Java source code and produces the runnable artifact (binary). You can download Maven from the Apache website (<https://maven.apache.org/download.cgi>). Follow the installation instructions in <https://maven.apache.org/install.html> to install Maven on your operating system. We use Maven version 3.5 to test all the samples.

INSTALLING CURL

Download and install the curl command line tool from the curl website (<https://curl.haxx.se/download.html>). You use curl in the book as a client application to access microservices. Most of the operating systems do have curl installed out of the box.

INSTALLING THE GIT COMMAND-LINE TOOL

Download and install the Git command-line client on your computer based on your operating system. You only use Git client once to clone our samples Git repository. It's not must to install the Git client, you can also download the complete sample Git repository as a zip file from here: <https://github.com/microservices-security-in-action/samples>. When you click on **Clone or download** button, you will find a link to download a zip file.

2.1.2 Clone samples repository

Once you complete the steps in section 2.1.1, you can clone the samples Git repository with the following command. Once successfully executed, it creates a directory called samples in your local file system, with all the samples we have for the book.

```
\> git clone https://github.com/microservices-security-in-action/samples.git
```

2.1.3 Compiling the Order Processing microservice

Once you complete the steps above, it's time to get your hands dirty and run your first microservice. First, open the command-line tool in your operating system, and navigate to the location on your file system to where you cloned the samples repository, and in the rest of the book, we identify this location as [samples].

```
\> cd [samples]/chapter02/sample01
```

Inside chapter02/sample01 directory, you'll find the source code corresponding to the Order Processing microservice. From within that directory, execute the following command to build the microservice:

```
\> mvn clean install
```

If you run into problems while running the command, it could be that old and incompatible dependencies reside in your local Maven repository. To get rid of such problems, try removing

(renaming) the `.m2` directory that resides in your home directory (`~/.m2/`). The above command instructs Maven to compile your source code and produce a runnable artifact known as a `.jar` file. Note that you need to have Java and Maven installed to execute this step successfully. If your build is successful, you see the message `BUILD SUCCESS`. If this is the first time you're using Maven to build a Spring Boot project, Maven downloads all the Spring Boot dependencies from their respective repositories; therefore, an Internet connection is required in the build phase. The first-time build is expected to take slightly longer than the next attempts. After the first build, Maven installs all the dependencies in your local file system, and that takes the build times down considerably, for the subsequent attempts.

If the build is successful, you should see a directory named `target` within your current directory. The `target` directory should contain a file named `com.manning.mss.ch02.sample01-1.0.jar` (Other files will be within the `target` directory, but you're not interested in them at the moment.) Then run the following command from `chapter02/sample01/` directory to spin up the Order Processing microservice. Here, we use a Maven plugin called, `spring-boot`:

```
\> mvn spring-boot:run
```

If the microservice started successfully, you should see a bunch of messages being printed on the terminal. At the bottom of the message stack, you should see this message:

```
Started OrderApplication in <X> seconds
```

By default Spring Boot starts the microservice on HTTP port 8080. If you have any other services running in your local machine, on port 8080, make sure to stop them, or else you can change the default port of the Order Processing microservice by changing the value of `server.port` property as appropriate in the `chapter02/sample01/src/main/resources/application.properties` file. But, then again it would be much easier to follow rest of the samples in the chapter, with minimal changes, if you keep the Order Processing microservice running on the default port.

2.1.4 Accessing the Order Processing microservice

By default, Spring Boot runs an embedded Apache Tomcat web server that listens for HTTP requests on port 8080. In this section, you access your microservice using `curl` as the client application. In case you run the Order Processing microservice on a custom port, make sure to replace the value of the port (8080) in the following command with the one you used. To invoke the microservice open your command-line client, and execute the following `curl` command:

```
\> curl -d "[{\\"items\": [{\"itemCode\": \"IT0001\", \"quantity\": 3}, {\"itemCode\": \"IT0004\", \"quantity\": 1}], \"shippingAddress\": \"No 4, Castro Street, Mountain View, CA, USA\"}]" -H \"Content-Type: application/json\" http://localhost:8080/orders
```

You should see this message on your terminal:

```
{"orderId": "d58uy556-7c9h-3j60-045j-
```

```
84756ht3t736","items":[{"itemCode":"IT0001","quantity":3},{"itemCode":"IT0004","quantity":1}],"shippingAddress":"No 4, Castro Street, Mountain View, CA, USA"}
```

If you see this message, you've successfully developed, deployed, and tested your first microservice!

NOTE All samples in this chapter use HTTP (not HTTPS) endpoints to spare you from having to set up proper certificates and to make it possible for you to inspect messages being passed on the wire (network), if required. In production systems, we do not recommend you to use HTTP for any endpoint. You should only expose all the endpoints over HTTPS. In chapter 6 we discuss how to secure microservices with HTTPS.

When you executed the above command, curl initiated an HTTP POST request to the /orders resource located on the server localhost on port 8080 (local machine). The content (payload) of the request represents an order placed for two items to be shipped to a particular address. The Spring Boot server runtime (embedded Tomcat) dispatched this request to the placeOrder function (in the Java code) of your Order Processing microservice, which responded with the message.

2.1.5 What is inside the source code directory?

Let's navigate inside sample01 directory and inspect its contents. You should see a file named pom.xml and a directory named src. Navigate to the src/main/java/com/manning/mss/ch02/sample01/service/ directory. You see two files: one named OrderApplication.java and the other named OrderProcesingService.java.

Before you dig into the contents of these files, let us explain what you're trying to build here. If you recall, a *microservice* is a collection of network-accessible functions. In this context, *network-accessible* means that these functions are accessible over the HTTP protocol (<https://tools.ietf.org/html/rfc2616>) through applications such as web browsers, mobile applications, or software such as curl (<https://curl.haxx.se/>) that's capable of communicating over the HTTP protocol. Typically, a function in a microservice is exposed as an action over a REST resource (<https://spring.io/understanding/REST>). Often, a resource represents an object or entity that you intend to inspect or manipulate. When mapped to HTTP, a resource is usually identified by a Request-URI, and an action is represented by an HTTP method (see sections 5.1.1 and 5.1.2 of the HTTP specification or the RFC2616 (<https://tools.ietf.org/html/rfc2616#page-35>)).

Consider a scenario in which an ecommerce application uses a microservice to retrieve the details of an order. An HTTP request template that maps to that particular function in the microservice looks similar to the following.

```
GET /orders/{orderid}
```

GET is the HTTP method used in this case since you're performing a data retrieval operation. /orders/{orderid} is the resource path on the server that hosts the corresponding microservice. This path can be used to uniquely identify an order resource. {orderid} is a

variable that needs to be replaced with proper values in the actual HTTP request. Something like GET /orders/d59dbd56-6e8b-4e06-906f-59990ce2e330 would ask the microservice to retrieve details of the order with id d59dbd56-6e8b-4e06-906f-59990ce2e330.

2.1.6 Understanding the source code of the microservice

Now that you have a fair understanding of how to expose a microservice as an HTTP resource, look at the code samples to see how to develop a function in Java and use Spring Boot to expose it as an HTTP resource. Use the file browser in your operating system to open the directory located at sample01/src/main/java/com/manning/mss/ch02/sample01/service, and open the OrderProcessingService.java file in a text editor. If you're familiar with Java IDEs (development environments) such as Eclipse, Netbeans, IntelliJ IDEA, or anything similar, you can import the sample as a Maven project to the IDE. Listing 2.1 shows what the content of the OrderProcessingService.java file looks like.

Listing 2.1. The content of the OrderProcessingService.java file

```
@RestController #A
@RequestMapping("/orders") #B
public class OrderProcessingService {

    private Map<String, Order> orders = new HashMap<>();

    @PostMapping #C
    public ResponseEntity<Order> placeOrder(@RequestBody Order order) {

        System.out.println("Received Order For " + order.getItems().size() + " Items");
        order.getItems().forEach((lineItem) -> System.out.println("Item: " +
lineItem.getItemCode() +
            " Quantity: " + lineItem.getQuantity()));

        String orderId = UUID.randomUUID().toString();
        order.setOrderId(orderId);
        orders.put(orderId, order);
        return new ResponseEntity<Order>(order, HttpStatus.CREATED);
    }
}
```

#A Informs the Spring Boot runtime that you're interested in exposing this class as a microservice

#B Specifies the path under which all the resources of the service exist

#C Informs the Spring Boot runtime to expose this function as a POST HTTP method

This code is a simple Java class with a function named placeOrder. As you may notice, we decorated the class with the @RestController annotation to inform the Spring Boot runtime that you're interested in exposing this class as a microservice. The @RequestMapping annotation specifies the path under which all the resources of the service exist. We also decorated the placeOrder function with the @PostMapping annotation, which informs the Spring Boot runtime to expose this function as a POST HTTP method (action) on the /orders context. The @RequestBody annotation says that the payload in the HTTP request is to be assigned to an object of type Order.

Another file within the same directory is named `OrderApplication.java`. Open this file with your text editor, and inspect its contents, which looks like as listed below.

```
@SpringBootApplication
public class OrderApplication {
    public static void main(String args[]) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

This simple Java class has only the `main` function. The `@SpringBootApplication` annotation informs the Spring Boot runtime that this application is a Spring application. It also makes the runtime check for `Controller` classes (such as the `OrderProcessingService` class you saw earlier) within the same package of the `OrderApplication` class. The `main` function is the function invoked by the Java Virtual Machine (JVM) when you command it to run the particular Java program. Within the `main` function, start the Spring Boot application through the `run` utility function of the `SpringApplication` class, which resides within the Spring framework.

2.2 Setting up an OAuth 2.0 server

Now that you have your first microservice up and running, we can start getting to the main focus of this book: securing microservices. You'll be using OAuth2.0 to secure your microservice at the edge. If you are unfamiliar with OAuth 2.0, we recommend you go through the appendix D first. Appendix D provides a comprehensive overview of OAuth 2.0 protocol and how it works. In chapter 3, we discuss in detail why we opted for OAuth2.0 over options such as basic authentication and certificate-based authentication. For now, know that OAuth 2.0 is a clean mechanism for solving the problems related to providing your username and password to an application that you don't trust to access your data. When combined with JSON Web Token (JWT)¹, OAuth2.0 can be a highly scalable authentication and authorization mechanism, which is critical when it comes to securing microservices. Those who know about OAuth 2.0 probably are raising their eyebrows at seeing it mentioned as a way of authentication. We agree that it's not an authentication protocol at the client application end, but at the resource server end, which is the microservice.

¹ A JSON Web Token (JWT) is a container that carries different types of assertions or claims from one place to another in a cryptographically safe manner. If you are new to JWT please check appendix H.

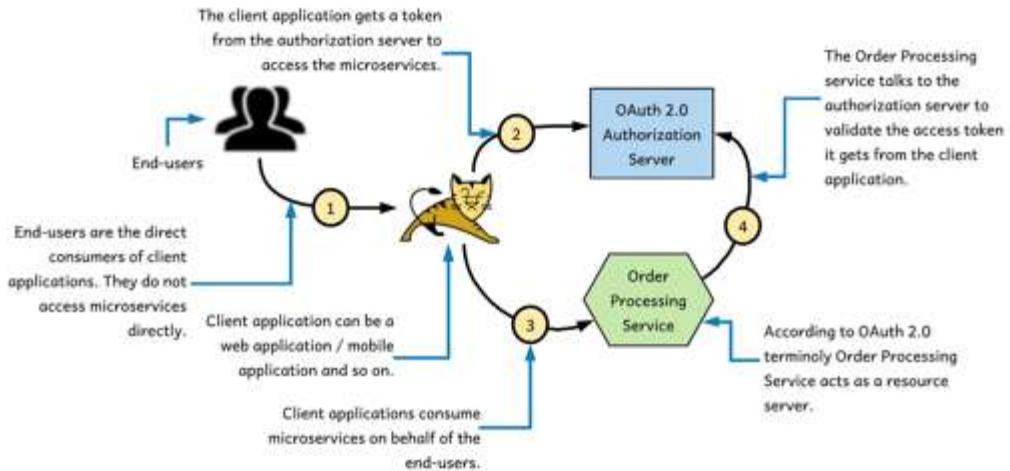


Figure 2.2 Actors in an OAuth2.0 flow. In a typical access delegation flow, a client accesses a resource that is hosted on a resource server on behalf of the end-user with a token provided by the authorization server.

2.2.1 The interactions with an authorization server

In an OAuth 2.0 flow, the client application, the end-user and the resource server all interact directly with the authorization server, in different phases (see figure 2.2). Before requesting a token from an authorization server, the client applications have to register themselves with it. Or in other words, an authorization server issues tokens only for the client applications it knows. Some authorization servers support Dynamic Client Registration Protocol (<https://tools.ietf.org/html/rfc7591>), which allows clients to register themselves on the authorization server on the fly – or on demand (see figure 2.3).

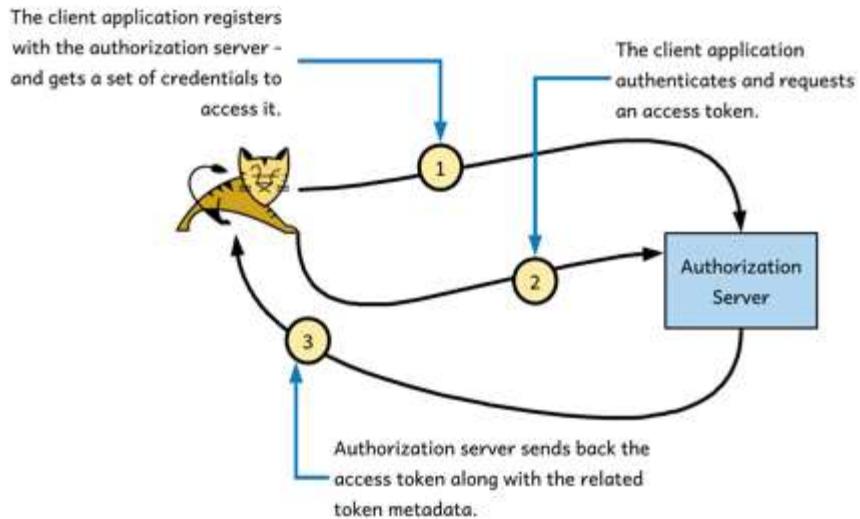


Figure 2.3: A client application is requesting an access token from the authorization server. The authorization server only issues tokens to known client applications. A client application must register at the authorization server first.

The Order Processing microservice, which plays the role of the resource server here, would receive the token issued by the authorization server, from the client, usually as an HTTP header or as a query parameter when the client makes an HTTP request (see step-1 in figure 2.4). It's recommended that the client communicates with the microservice over HTTPS and send the token in an HTTP header instead of a query parameter. Because query parameters are sent in the URL, those can be recorded in server logs. Hence, anyone who has access to the logs can see this information.

Having Transport Layer Security (TLS) to secure the communication (or in other words, the use of HTTPS), between all the entities in an OAuth 2.0 flow is extremely important. The token (access token), the authorization server issues, to access a microservice (or a resource) must be protected like a password. We do not send passwords over plain HTTP and always use HTTPS. Hence we follow the same when sending access tokens over the wire.

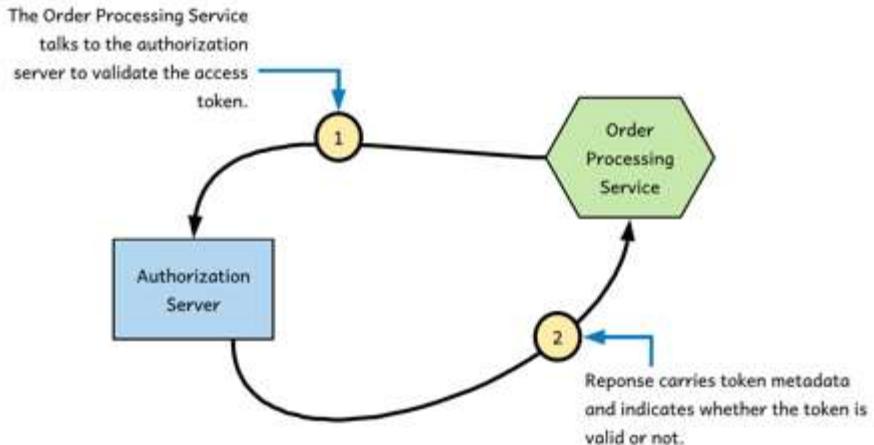


Figure 2.4: A client application is passing the OAuth access token in the HTTP authorization header to access a resource from the resource server.

Upon receipt of the access token, the Order Processing microservice should validate it against the authorization server before granting access to its resources. An OAuth 2.0 authorization server usually supports the OAuth 2.0 Token Introspection Profile (<https://tools.ietf.org/html/rfc7662>) or a similar alternative for resource servers to check the validity of an access token (see figure 2.5). If the access token is a self-contained JWT, then the resource server can validate it, by itself, without talking to the authorization server. We discuss self-contained JWT in detail, in chapter 6.

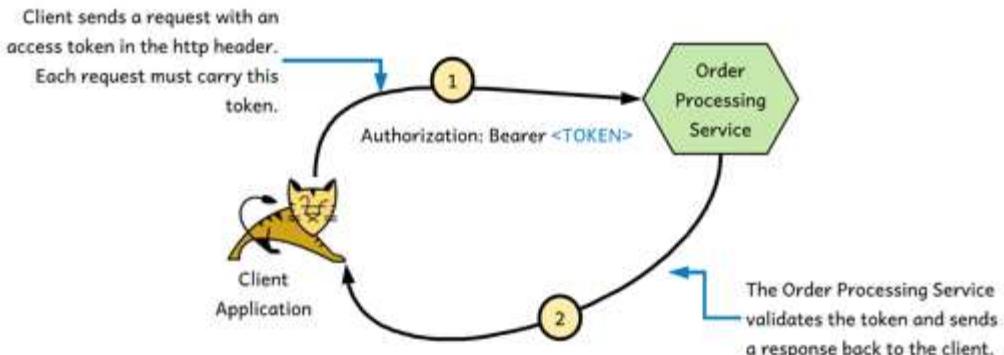


Figure 2.5: The Order Processing microservice (resource server) introspects the access token by talking to the authorization server.

2.2.2 Running the OAuth 2.0 authorization server

There are many production-grade OAuth 2.0 authorization servers out there, both proprietary and open source. However, in this chapter we use a simple authorization server that's capable of issuing access tokens. It is built using Spring Boot. Within the Git repository you cloned earlier, you should find a directory named `sample02` under the directory `chapter02`. There you find the source code of the simple OAuth 2.0 server. First, compile and run it; then look into the code to understand what it does.

To compile, use your command-line client to navigate into the `chapter02/sample02` directory. From within that directory, execute the following Maven command to compile and build the runnable artifact:

```
\> mvn clean install
```

If your build is successful, you see the message `BUILD SUCCESS`. You should find a file named `com.manning.mss.ch02.sample02-1.0.jar` within a directory named `target`. Execute the following command from within the `chapter02/sample02` directory, using your command-line client, to run the OAuth 2.0 server:

```
\> mvn spring-boot:run
```

If you managed to run the server successfully, you should see this message:

```
Started OAuthServerApplication in <X> seconds
```

This message indicates that you successfully started the authorization server. By default OAuth server runs on HTTP port 8085. If you have any other services running in your local machine, on port 8085, make sure to stop them, or else you can change the default port of the authorization server by changing the value of `server.port` property as appropriate in the `chapter02/sample02/src/main/resources/application.properties` file. But, then again it would be much easier to follow rest of the samples in the chapter, with minimal changes, if you keep the authorization server running on the default port.

NOTE The authorization server used in this chapter is running on HTTP, while in a production deployment it must be over HTTPS. In chapter 6, we discuss how to set up an authorization server over HTTPS.

2.2.3 Getting an access token from the OAuth 2.0 authorization server

To get an access token from the authorization server, use an HTTP client to make an HTTP request to the server. In the real world, the client application that is accessing the microservice would make this request. You'll be using curl for this purpose as the HTTP client. To request an access token from the authorization server (which runs on port 8085), run the following command, using your command-line client:

```
\> curl -u orderprocessingapp:orderprocessingappsecret -H "Content-Type: application/json" -d
  '{ "grant_type": "client_credentials", "scope": "read write" }'
  'http://localhost:8085/oauth/token'
```

Take a quick look at this request, and try to understand it. You can think of `orderprocessingapp:orderprocessingappsecret` as the client application's username (`orderprocessingapp`) and password (`orderprocessingappsecret`). The only difference is that these credentials belong to an application, not a user. The application being used to request a token needs to bear a unique identifier and a secret that's known by the authorization server. The `-u` flag provided to curl instructs it to create a basic authentication header and send it to the authorization server as part of the HTTP request. Then curl base64-encodes the `orderprocessingapp:orderprocessingappsecret` string and creates the Basic authentication HTTP header as follows:

```
Authorization: Basic b3JkZXJwcm9jZXNzaW5nYXBwOm9yZGVycHJvY2Vzc2luZ2FwcHN1Y3J1dA==
```

The string that follows the `Basic` keyword is the base64-encoded value of `orderprocessingapp:orderprocessingappsecret`. As you may have noticed, you're sending a Basic authentication header to the token endpoint of the OAuth2.0 server because the token endpoint usually is protected through Basic authentication (<https://tools.ietf.org/html/rfc2617>). Because the client application is requesting a token here, the Basic authentication header should consist of the credentials of the client application, not of a user. Note that Basic authentication here isn't used for securing the resource server (or the microservice); you use OAuth2.0 for that purpose. Basic authentication at this point is used only for obtaining the OAuth token required to access the microservice, from the authorization server. In chapter 3, we discuss in detail why we chose OAuth over protocols such as Basic authentication and TLS to secure your resource server. Even for securing the token endpoint of the OAuth server, instead of Basic authentication, you can pick whatever the authentication mechanism you prefer. For strong authentication, many prefer using certificates.

The parameter `-H "Content-Type: application/json"` in the above token request, informs the authorization server that the client will be sending a request in JSON format. What follows the `-d` flag is the actual JSON content of the message. In the JSON message, the `grant_type` specifies the protocol to be followed in issuing the token. We talk more about OAuth grant types in chapter 3. For now, think of a grant type as the sequence of steps that the client application and the authorization server follow to issue an access token. In the case of `client_credentials` grant type, the authorization validates the Basic authentication header and issues an access token if it's valid.

The `scope` declares what actions the application intends to perform with a token. When issuing a token, the authorization server validates whether the requesting application is permitted to obtain the requested scopes and binds them to the token as appropriate. If the application identified by `orderprocessingapp` can only do read operations, for example, the authorization server issues the corresponding token under the scope, `read`. The URL `http://localhost:8085/oauth/token` is the endpoint of the authorization server that issues access tokens. Your curl application sends the HTTP request to this endpoint to obtain an access token. If your request is successful, you should see a response similar to this:

```
{"access_token":"8c017bb5-f6fd-4654-88c7-
```

```
c26cccc54bdd", "token_type": "bearer", "expires_in": 300, "scope": "read write"}
```

2.2.4 Understanding the access token response

Following lists down the details on the JSON response from the authorization server in section 2.2.4. If you are new to OAuth 2.0, please check appendix D for further details.

- `access_token`—The value of the token issued by the authorization server to the client application (`curl`, in this case).
- `token_type`—The token type (more about this topic when we talk about OAuth 2.0 in appendix D). Most of the OAuth deployments we see today are using bearer tokens.
- `expires_in`—The period of validity of the token, in seconds. The token would be considered to be invalid (expired) after this period.
- `scope`—The actions that the token is permitted to perform on the resource server (microservice).

2.3 Securing a microservice with OAuth2.0

So far, you've learned how to develop your first microservice and how to set up an authorization server to get an access token. In this section, you see how to secure the microservice you developed. Up to now, you've accessed it without any security in place.

2.3.1 Security based on OAuth2.0

In this section, you see how to use OAuth2.0 to secure the Order Processing microservice. The secured microservice now expects a valid security token (access token) from the calling client application. Then it validates this access token with the assistance of the authorization server before it grants access to its resources. Figure 2.6 illustrates this scenario.

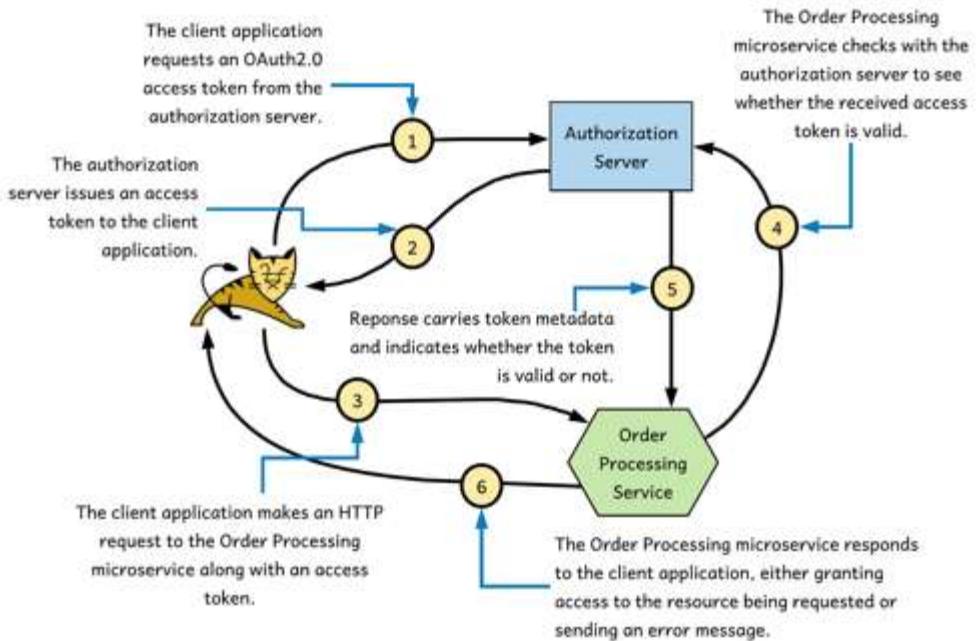


Figure 2.6: A client application accessing a secured microservice with an access token obtained from the authorization server. The Order Processing microservice talks to the authorization server to validate the token, before granting access to its resources.

Here's what happens in each of the steps illustrated in figure 2.6:

- *Step 1*—Client application requests an OAuth2.0 access token from the authorization server.
- *Step 2*—In response to the request in step 1, the authorization server issues an access token to the client application.
- *Step 3*—The client application makes an HTTP request to the Order Processing microservice. This request carries the access token obtained in step 2 as an HTTP header.
- *Step 4*—The Order Processing microservice checks with the authorization server to see whether the received access token is valid.
- *Step 5*—In response to the request in step 4, the authorization server checks to see whether the provided access token is an active token in the system and whether it is valid for that particular moment (it isn't expired). Then it responds to the Order Processing microservice, indicating whether the access token is valid.
- *Step 6*—In response to the request on step 3, based on the result in step 5, the Order Processing microservice responds to the client application, either granting access to the resource being requested or sending an error message.

resource being requested or sending an error message.

In the examples in this chapter so far, you used the `client_credentials` grant type to obtain an access token from the authorization server. In this particular case, the token endpoint of the authorization server is protected via Basic authentication with the client ID and the client secret of the application. The `client_credentials` grant type is good in cases where the client application needs not to worry about end-users. In case it has to, then it should pick an appropriate grant type. Mostly the `client_credentials` grant type is used for system-to-system authentication.

2.3.2 Running the sample

If you're still running the Order Processing microservice from section 2.1, stop it, because you're about to start a secured version of the same microservice on the same port. You can stop the microservice by going to the terminal window that is running it and pressing Ctrl-C (Command-C on a Mac). To run this sample, navigate to the directory where you cloned the samples from the Git repository from your command-line application, and go to the `chapter02/sample03` directory. From within that directory, execute the following Maven command to build the sample:

```
\> mvn clean install
```

If the build is successful, you should see a directory named `target` within your current directory. The `target` directory should contain a file named `com.manning.mss.ch02.sample03-1.0.jar` (Other files will be within the `target` directory, but you're not interested in them at the moment.) Then run the following command from `chapter02/sample03/` directory to spin up the secured Order Processing microservice. Here, we use a Maven plugin called, `spring-boot`:

```
\> mvn spring-boot:run
```

If you managed to run the server successfully, you should see a message like this:

```
Started OrderApplication in <X> seconds
```

Now run the same curl command you used earlier in this chapter to access the Order Processing microservice:

```
\> curl -d "[{"items": [{"itemCode": "IT0001", "quantity": 3}, {"itemCode": "IT0004", "quantity": 1}], "shippingAddress": "No 4, Main Street, Colombo 1, Sri Lanka"}]" -H "Content-Type: application/json" http://localhost:8080/orders
```

You should see an error message saying that the request was unsuccessful. The expected response message is

```
{"error": "unauthorized", "error_description": "Full authentication is required to access this resource"}
```

Your Order Processing microservice is now secured and can no longer be accessed without a valid access token obtained from the authorization server. To understand how this happened, look at the modified source code of your Order Processing microservice. Using your favorite text editor or IDE, open the file `OrderProcessingService.java` located inside `src/main/java/com/manning/mss/ch02/sample03/service` directory. This is more or less the same class file you inspected earlier with a function named `placeOrder`. One addition to this class is the annotation `@EnableWebSecurity`. This annotation informs your Spring Boot runtime to apply security to the resources of this microservice. Following is the class definition:

```
@EnableResourceServer
    @EnableWebSecurity
    @RestController
    @RequestMapping("/orders")
    public class OrderProcessingService extends WebSecurityConfigurerAdapter {
```

If you further inspect this class, you should notice a method named `tokenServices` that returns an object of type `ResourceServerTokenServices` (see listing 2.2) Properties set in the `RemoteTokenServices` object (which is of the `ResourceServerTokenServices` type) are the ones that the Spring Boot runtime uses to communicate with the authorization server to validate credentials received by the Order Processing microservice (the resource server, in this case). If you go through the code of the `tokenServices` function, you see that it uses a method named `setCheckTokenEndpointUrl` to set the value `http://localhost:8085/oauth/check_token` as the `TokenEndpointURL` property in the `RemoteTokenServices` class. The `TokenEndpointURL` property is used by the Spring Boot runtime to figure out the URL on the OAuth server that it has to talk to validate any tokens it receives via HTTP requests. This is the URL the Order Processing microservice uses in step 4 of figure 2.6 in section 2.3 to talk to the authorization server.

Listing 2.2 The `tokenService()` method from `OrderProcessingService.java`

```
@Bean
public ResourceServerTokenServices tokenServices() {
    RemoteTokenServices tokenServices = new RemoteTokenServices();
    tokenServices.setClientId("orderprocessingservice");
    tokenServices.setClientSecret("orderprocessingservicesecret");
    tokenServices.setCheckTokenEndpointUrl("http://localhost:8085/oauth/check_token");
    return tokenServices;
}
```

The endpoint that does the validation of the token itself is secure; it requires a valid Basic authentication header. This header should consist of a valid client ID and a client secret. In this case, one valid client ID and client secret pair is `orderprocessingservice` and `orderprocessingservicesecret`, which is why those values are set in the `RemoteTokenServices` object. In fact these credentials are hard coded in the simple OAuth server we developed.

In section 2.4, you see how to use the token you obtained from the authorization server in section 2.2 to make a request to the now-secure Order Processing microservice.

2.4 Invoking a secured microservice from a client application

Before a client application can access your secured Order Processing microservice, it should obtain an OAuth2.0 access token from the authorization server. As explained in section 2.2.4, the client application at minimum requires a valid client ID and a client secret to obtain this token. The client ID and client secret registered on your OAuth server at the moment are `orderprocessingapp` and `orderprocessingappsecret`, respectively. As before, you can use the following curl command to obtain an access token:

```
\> curl -u orderprocessingapp:orderprocessingappsecret -H "Content-Type: application/json" -d '{ "grant_type": "client_credentials", "scope": "read write" }' 'http://localhost:8085/oauth/token'
```

If the request is successful, you should get an access token in response, as follows:

```
{"access_token":"8c017bb5-f6fd-4654-88c7-c26ccca54bdd","token_type":"bearer","expires_in":300,"scope":"read write"}
```

As discussed earlier, `8c017bb5-f6fd-4654-88c7-c26ccca54bdd` is the value of the access token you got, and it's valid for 5 minutes (300 seconds). This access token needs to be provided to the HTTP request you'll make to the Order Processing microservice. You need to send the token as an HTTP header named `Authorization`, and the header value needs to be prefixed by the string `Bearer`, as follows:

```
Authorization: Bearer 8c017bb5-f6fd-4654-88c7-c26ccca54bdd
```

The new curl command to access the order microservice would be:

```
\> curl -d "{\"items\": [{\"itemCode\": \"IT0001\", \"quantity\": 3}, {\"itemCode\": \"IT0004\", \"quantity\": 1}], \"shippingAddress\": \"No 4, Main Street, Colombo 1, Sri Lanka\"}" -H "Content-Type: application/json" -H "Authorization: Bearer 8c017bb5-f6fd-4654-88c7-c26ccca54bdd" http://localhost:8080/orders
```

Note that the `-H` parameter is used to pass the access token as an HTTP header named `Authorization`. This time, you should see the Order Processing microservice responding with a proper message saying that the order was successful:

```
{"orderId":"d59dbd56-6e8b-4e06-906f-59990ce2e330","items":[{"itemCode":"IT0001","quantity":3},{"itemCode":"IT0004","quantity":1}],"shippingAddress":"No 4, Main Street, Colombo 1, Sri Lanka"}
```

If you see this message, you've successfully created, deployed, and tested a secured microservice. Congratulations! The access token that the client application (curl) sent in the HTTP header to the Order Processing microservice was validated against the authorization server. This process is called token *introspection*. Because the result of the introspection operation ended up being a success, the Order Processing microservice granted access to its resources.

2.5 Service-level authorization with OAuth 2.0 scopes

You need a valid access token to access a microservice. Authentication is the first level of defense applied to a microservice to protect it from spoofing. The authentication step that occurs before granting access to the microservice ensures that the calling entity is a valid client (user, application, or both) in the system. Authentication, however, doesn't mention anything about the level of privileges the client has in the system.

A given microservice may have more than one operation. The Order Processing microservice, for example, has one operation for creating orders (`POST /orders`) and another operation for retrieving order details (`GET /orders/{id}`). Each operation in a microservice may require a different level of privilege for access.

A *privilege* describes what actions you're permitted to perform on a resource. More often than not, your role(s) in an organization describe(s) what actions you're permitted to perform within the organization and what actions you're not permitted to perform. A privilege may also indicate status or credibility. If you've traveled on a commercial airline, you're familiar with or have heard of the membership status of travelers who belong to the airlines' frequent-flyer programs. Likewise, a privilege is an indication of the level of access that a user or an application possesses in a system.

In the world of OAuth2.0, privilege is mapped to a scope. A *scope* is way of abstracting a privilege. A privilege can be a user's role, membership status, credibility, or something else. It can also be a combination of a few such attributes. You use scopes to abstract the implication of a privilege. A scope declares the privilege required by a calling client application to grant access to a resource. The `placeOrder` operation, for example, requires a scope called `"write"`, and the `getOrder` operation requires a scope called `"read"`. The implications of `"write"` and `"read"`—whether they're related to user roles, credibility, or anything else—is orthogonal from a resource-server point of view.

2.5.1 Obtaining a scoped access token from the authorization server

The authorization server you built in this chapter contains two applications: one with client ID `orderprocessingapp`, which you used for accessing the microservice, and one with client ID `orderprocessingservice`. You configured these applications in such a way that the first application, with client ID `orderprocessingapp`, has privileges to obtain both scopes `"read"` and `"write"`, whereas the second application, with client ID `orderprocessingservice`, has privileges to obtain only scope `"read"`. Listing 2.3 explains.

Listing 2.3. The `configure()` method in `OAuthServerConfig.java` from sample02

```
clients.inMemory()
.withClient("orderprocessingapp").secret("orderprocessingsecret")
.authorizedGrantTypes("client_credentials", "password")
.scopes("read", "write")
.accessTokenValiditySeconds(3600)
.resourceIds("sample-oauth")
.and()
.withClient("orderprocessingservice").secret("orderprocessingservicesecret")
.authorizedGrantTypes("client_credentials", "password")
```

```
.scopes("read")
.accessTokenValiditySeconds(3600)
.resourceIds("sample-oauth");
```

This code (see listing 2.3) means that anyone who uses `orderprocessingapp` is allowed to obtain an access token under both scopes "read" and "write", whereas any user of `orderprocessingservice` is allowed to obtain an access token only under scope "read". In the request to obtain an access token, you used `orderprocessingapp` as the client ID and requested both scopes "read" and "write". Now execute the same request to obtain an access token with `orderprocessingservice` as the client ID to see what the token response looks like. Execute this curl command to make the token request:

```
\> curl -u orderprocessingservice:orderprocessingservicesecret -H "Content-Type: application/json" -d '{ "grant_type": "client_credentials", "scopes": "read write" }' 'http://localhost:8085/oauth/token'
```

If the token request was successful, you should see this response:

```
{"access_token":"47190af1-624c-48a6-988d-f4319d36b7f4","token_type":"bearer","expires_in":3599,"scope":"read"}
```

Notice that although in the token request, you requested both scopes "read" and "write", the OAuth server issued a token with scope "read" only. One good thing about the OAuth server is that although you may not have the privileges to get all the scopes you request, instead of refusing to issue an access token, the server issues an access token bound to the scopes that you're entitled to. Then again, this may vary based on the OAuth server you pick – and the OAuth 2.0 standard does not mandate a way how the OAuth servers should handle such cases.

2.5.2 Protecting access to a microservice with OAuth 2.0 scopes

Now you have an idea of how an authorization server grants privileges to a token based on the scopes. In this section, you see a how the resource server or the microservice enforces these scopes on the resources it wants to protect. Listing 2.4 (`chapter02/sample03/src/main/java/com/manning/mss/ch02/sample03/service/ResourceServerConfig.java` class file) explains how the resource server enforces these rules.

Listing 2.4 The code from ResourceServerConfig.java

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

    private static final String SECURED_READ_SCOPE =
        "#oauth2.hasScope('read')";

    private static final String SECURED_WRITE_SCOPE =
        "#oauth2.hasScope('write')";

    private static final String SECURED_PATTERN_WRITE = "/orders/**";
```

```

private static final String SECURED_PATTERN_READ = "/orders/{id}";

@Override
public void configure(HttpSecurity http) throws Exception {

    http.requestMatchers()
        .antMatchers(SECURED_PATTERN_WRITE).and().authorizeRequests()
            .antMatchers(HttpMethod.POST,
                SECURED_PATTERN_WRITE).access(SECURED_WRITE_SCOPE)
            .antMatchers(HttpMethod.GET,
                SECURED_PATTERN_READ).access(SECURED_READ_SCOPE);

}

@Override
public void configure(ResourceServerSecurityConfigurer resources) {
    resources.resourceId("sample-oauth");
}
}

```

As you can see, the code instructs the microservice runtime (Spring Boot) to check for the relevant scope for the particular HTTP method and request path. This line of code

```
.antMatchers(HttpMethod.POST,
    SECURED_PATTERN_WRITE).access(SECURED_WRITE_SCOPE)
```

checks for the scope "write" for any POST request made against the request path that matches the regular expression /orders/**. Similarly, this line of code checks for the scope "read" for GET requests on path /orders/{id}:

```
.antMatchers(HttpMethod.GET,
    SECURED_PATTERN_READ).access(SECURED_READ_SCOPE)
```

Now try to access the POST /orders resource with the token that has only a "read" scope. Execute the same curl command you used last time to access this resource, but with a different token this time (one that has read access only):

```
\> curl -d "{\"items\": [{\"itemCode\": \"IT0001\", \"quantity\": 3}, {\"itemCode\": \"IT0004\", \"quantity\": 1}], \"shippingAddress\": \"No 4, Main Street, Colombo 1, Sri Lanka\"}" -H "Content-Type: application/json" -H "Authorization: Bearer 47190af1-624c-48a6-988d-f4319d36b7f4" http://localhost:8080/orders
```

When this command executes, you should see this error response from the resource server:

```
{"error": "insufficient_scope", "error_description": "Insufficient scope for this resource", "scope": "write"}
```

This response says that the token's scope for this particular operation is insufficient and that the required scope is "write".

Assuming that you still have a valid orderId (d59dbd56-6e8b-4e06-906f-59990ce2e330) from a successful request to the POST /orders operation, try to make a GET /orders/{id} request with the preceding token to see whether it's successful. You can use the following curl command to make this request. Note that the orderId used in the

example won't be the same `orderId` you got when you tried to create an order yourself. Use the one that you received instead of the one used in this example.

```
\> curl -H "Authorization: Bearer 47190af1-624c-48a6-988d-f4319d36b7f4"
      http://localhost:8080/orders/d59dbd56-6e8b-4e06-906f-59990ce2e330
```

This request should give you a successful response, as follows. The token that you obtained bears the "read" scope, which is what the `GET /order/{id}` resource requires, as declared on the resource server:

```
{"orderId": "d59dbd56-6e8b-4e06-906f-59990ce2e330", "items": [{"itemCode": "IT0001", "quantity": 3}, {"itemCode": "IT0004", "quantity": 1}], "shippingAddress": "No 4, Main Street, Colombo 1, Sri Lanka"}
```

Throughout this chapter, we've covered the most primary mechanism of securing a microservice and accessing a secured microservice. As you may imagine, this chapter is only the beginning. Real-world scenarios demand a lot more than an application with a valid client ID and client secret gaining access to resources on a microservice. We discuss all these options throughout the rest of this book.

2.6 Summary

- OAuth 2.0 is an authorization framework, which is widely used in securing microservice deployments at the edge.
- OAuth 2.0 supports multiple grant types and the client credentials grant type, which we used in this chapter, is mostly used for system-to-system authentication.
- Each access token issued by an authorization server is coupled with one or more scopes – and scopes are used in OAuth 2.0 to express the privileges attached to an access token.
- OAuth 2.0 scopes are used to protect and enforce access control checks in certain operations in microservices.

3

Securing north/south traffic with an API gateway

This chapter covers

- The role an API gateway in a microservices deployment
- The deficiencies in the architecture we followed in chapter 2 in securing a microservice, and how to improve
- Why OAuth 2.0 is the de facto standard for securing microservices at the edge
- How to deploy a microservice behind the Zuul API gateway and secure it with OAuth 2.0

In chapter 2, we discussed how to secure microservices at the edge with OAuth 2.0. The focus of chapter 2 was to get things started with a quite straightforward deployment. The samples there were far from production ready. Each microservice had to connect to an OAuth authorization server for token validation and decide which OAuth authorization server it wanted to trust. This is not a scalable model when you have hundreds of microservices and too much responsibility on the microservices developer. Please note that, when we say just OAuth, we in fact mean OAuth 2.0 – and if you are not familiar with OAuth, we recommend you go through the appendix D first.

In an ideal world, the microservices developer should only worry about the business functionality of a microservice, and the rest should be handled by specialized components with less hassle. The API Gateway and Service Mesh are two architectural patterns that help us reach there. In this chapter we discuss the API Gateway pattern and in chapter 12, about the Service Mesh pattern. The API Gateway pattern is mostly about edge security, while the Service Mesh pattern deals with service-to-service security. Or in other words the API Gateway deals with north/south traffic, while the Service Mesh deals with east/west traffic. We call the

software that implements the API Gateway pattern, an API gateway – and the software that implements the Service Mesh pattern, a service mesh.

Edge security is about protecting a set of resources (say for example a set of microservices) at the entry point to the deployment, at the API gateway. The API gateway is the only entry point to our microservices deployment for requests originating from outside. In a service mesh pattern, the architecture is much more decentralized. Each microservice has its own policy enforcement point as much as closer to the service – mostly it is a proxy, running next to each microservice. The API gateway is the centralized policy enforcement point to the entire microservices deployment, while in a service mesh a proxy running along with each microservice provides another level of policy enforcement at the service level. In chapter 12 we discuss how to leverage API Gateway pattern we discuss in this chapter, along with the Service Mesh pattern to build an end-to-end security solution.

3.1 The need for an API gateway in a microservices deployment

In this section we discuss the importance of having an API gateway in a microservices deployment. The API gateway is a crucial piece of infrastructure in our architecture since it plays a critical role that helps us clearly separating the functional requirements from the non-functional ones. We extend chapter 2's use case (a retail store) and look at a few problems in that use case and explain how we can solve them using the API Gateway pattern.

In a typical microservices deployment, microservices are not exposed directly to client applications. In most cases, microservices are behind a set of APIs that is exposed to the outside world via an API gateway. The API gateway is the entry point to the microservices deployment, which screens all incoming messages for security and other quality of service features. Figure 3.1 depicts a microservices deployment that resembles Netflix's, in which all the microservices are fronted by the Zuul API gateway. Zuul provides dynamic routing, monitoring, resiliency, security, and more. It acts as the front door to Netflix's server infrastructure, handling traffic from Netflix users around the world. In figure 3.1, Zuul is used to expose the Order Processing microservice via an API. We do not expose Inventory and Delivery microservices from the API gateway, because external applications need not to have access to those.

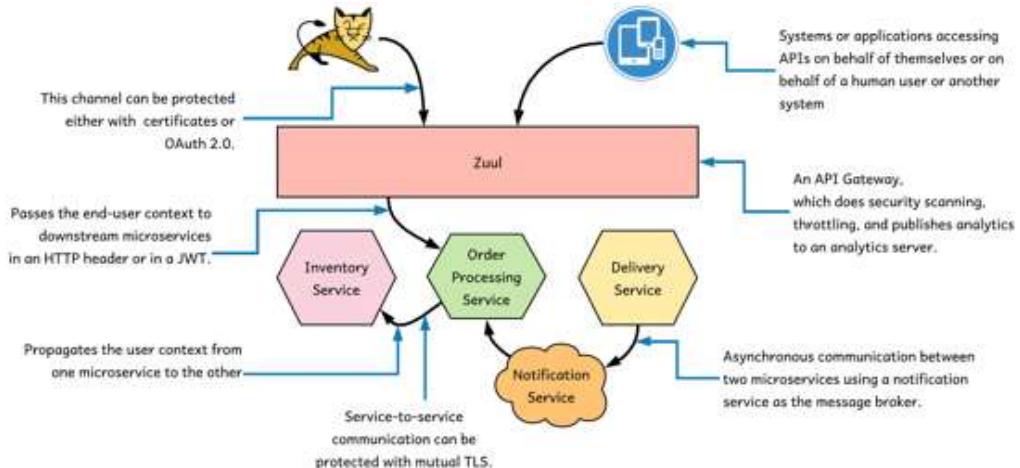


Figure 3.1 A typical microservices deployment with an API gateway. The API gateway screens all incoming messages for security and other quality of service features.

3.1.1 Decoupling security from the microservice

One key aspect of microservice best practices is the Single Responsibility Principle (https://en.wikipedia.org/wiki/Single_responsibility_principle). It is a principle used commonly in programming in which it states that every module, class or function should be responsible over a single part of the software's functionality. Under this principle, each microservice should be performing only one particular function. In the examples in chapter 2, the secured Order Processing microservice was implemented in such a way that it had to talk to the authorization server and validate the access tokens it got from client applications, in addition to the core business functionality of processing orders. As figure 3.2 shows, the Order Processing microservice had to worry about multiple tasks listed below.

1. Extracts the security header (token) from the incoming requests
2. Has to know beforehand the location of the authorization server it has to talk to validate the security token
3. Must be aware of the protocol and message formats to communicate with the authorization server to validate the security token
4. Gracefully handles errors in the token validation flow, because the microservice is directly exposed to the client application
5. Performs the business logic related to processing orders

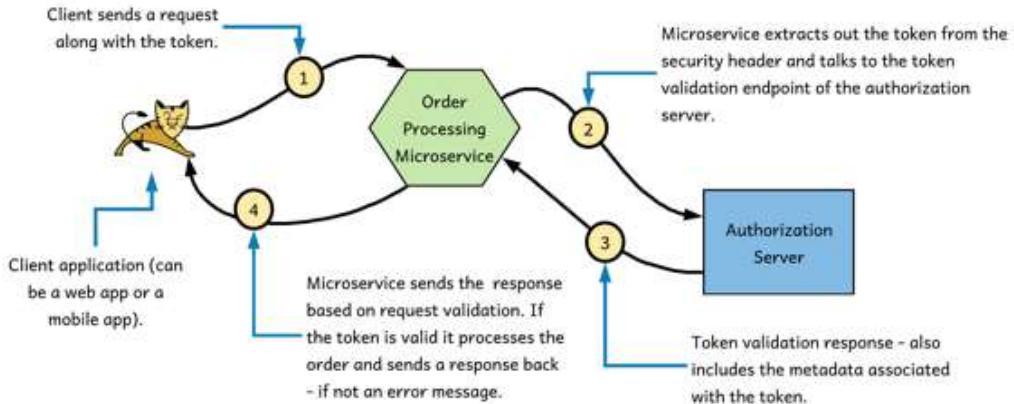


Figure 3.2 The interactions among the client application, microservice, and authorization server. The Order Processing microservice handles more functionality than ideally it should handle.

Executing all these steps becomes a problem because the microservice loses its atomic characteristics by performing more operations than it's supposed to. It would be ideal for the microservice to perform only the operation 5, which is the one that deals with the business logic for which we designed the microservice. The coupling of security and business logic introduces unwanted complexity and maintenance overhead to the microservice. For example, making changes in the security protocol would require changes in the microservice code and also scaling up the microservice would result in more connections to the authorization server.

CHANGES IN THE SECURITY PROTOCOL REQUIRES CHANGES IN THE MICROSERVICE

Someday, if you decide to move from OAuth 2.0 to something else as the security protocol enforced on the microservice, you have to make changes in the microservice, even though it may not have any changes related to its business logic. Also if you find a bug in the current security protocol, you need to patch the microservice code to fix it. This unwanted overhead compromises your agility in designing, developing, and deploying your microservice.

SCALING UP THE MICROSERVICE RESULTS IN MORE CONNECTIONS TO THE AUTHORIZATION SERVER

In certain cases, you need to run more instances of the microservice to cater to rising demand. Think of Thanksgiving, when people would be placing more orders in your retail store than usual, which would require you to scale up your microservice to meet the demand. Since each microservice talks to the authorization server for token validation, when you scale your microservice that will also increase the number of connections to the authorization server. There is a difference between 50 users using a single instance of a microservice and 50 users using 10 instances of a microservice. To cater to these 50 users, a single microservice may maintain a connection pool of about 5 to communicate with the authorization server. When

each instance of the microservice maintains a connection pool of 5 to connect to the authorization server, 10 instances of the microservice end up creating 50 connections on the authorization server as opposed to 5. Figure 3.3 is a scaled-down illustration of scaling up a microservice to meet increased demand.

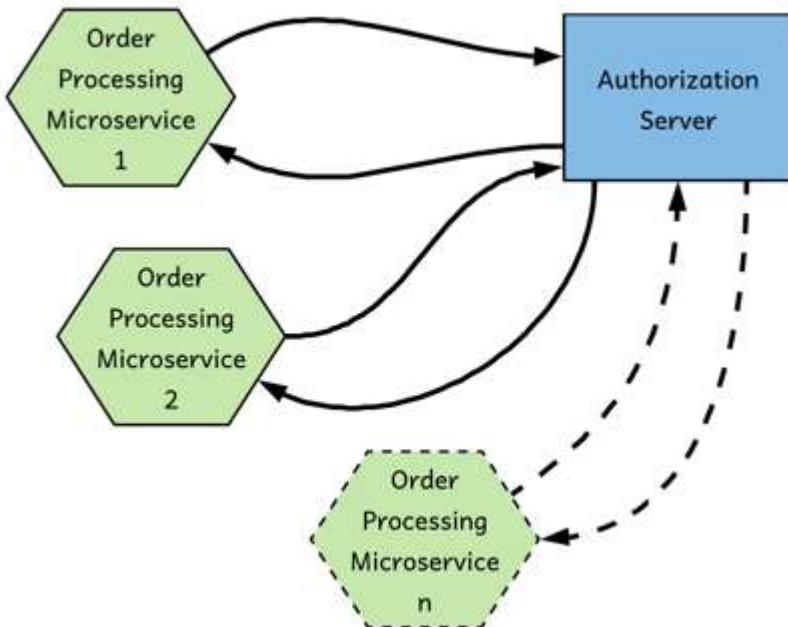


Figure 3.3 The effect on the authorization server when the microservice scales up, which results in more load on the authorization server.

An API gateway helps in decoupling security from a microservice. It intercepts all the requests coming to a microservice, talks to the respective authorization and dispatches only the legitimate requests to the upstream microservice. Otherwise, it returns back an error message to the client application.

3.1.2 The inherent complexities of microservice deployments make them harder to consume

A microservices deployment typically consists of many microservices and many interactions among these microservices (figure 3.4).

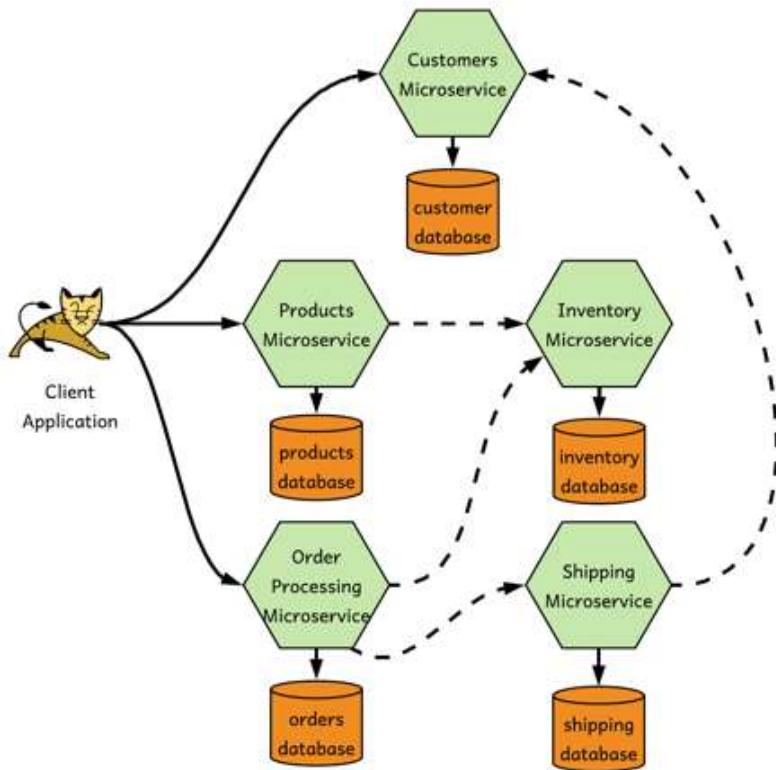


Figure 3.4 Architecture diagram of a microservices deployment, illustrating the services and connections among them.

As depicted in figure 3.4, an application that consumes microservices to build its own functionality must be capable of communicating with several microservices. Think of an organization with several teams, in which each team has the responsibility to develop one of the microservices shown in figure 3.4. Developers on each team could be using their own technology stacks for the microservices and using their own standards and practices. The nonuniformity of these microservices makes it hard for the consuming application because the developers of the consuming application need to learn how to work with many inconsistent interfaces. An API gateway solution, which usually comes as part of API management software, can be used to bring consistency to the interfaces that are being exposed to the consuming applications. The microservices themselves could be inconsistent, because they're now hidden from the outside world, and the API gateway can deal with the complications of interacting with the microservices.

3.1.3 The rawness of the microservices does not make them ideal for external exposure

Microservices can be as granular as they need to be. Suppose that you have two operations in your Products microservice: one for retrieving your product catalog and another for adding items to the catalog. From a REST point of view, the operation that retrieves the products would be modeled as `GET` on the `/products` resource, and the operation that adds products would be modeled as `POST` on the `/products` resource.

`GET /products` gets the list of products (read operation). `POST /products` adds a new product to the list of products (write operation).

In practice, you could expect more requests for the read operation than the write operation, because on a retail website, people browse for products much more frequently than items are added to the catalog. Therefore, you could decide to implement the `GET` and `POST` operations on two different microservices—maybe even on different technology stacks—so that they can scale out the microservices independently. This solution increases robustness because the failure of one microservice doesn't affect the operations performed by the other microservice. From a consuming point of view, however, it would be odd for the consuming applications to have to talk to two endpoints (two APIs) for the add and retrieve operations. A strong REST advocate could argue that it makes more sense to have these two operations on the same API (same endpoint).

The API Gateway architectural pattern is an ideal solution to this problem. It provides the consuming application a single API with two resources (`GET` and `POST`). Each resource can be backed by a microservice of its own, providing the scalability and robustness required by the microservices layer (see figure 3.5).

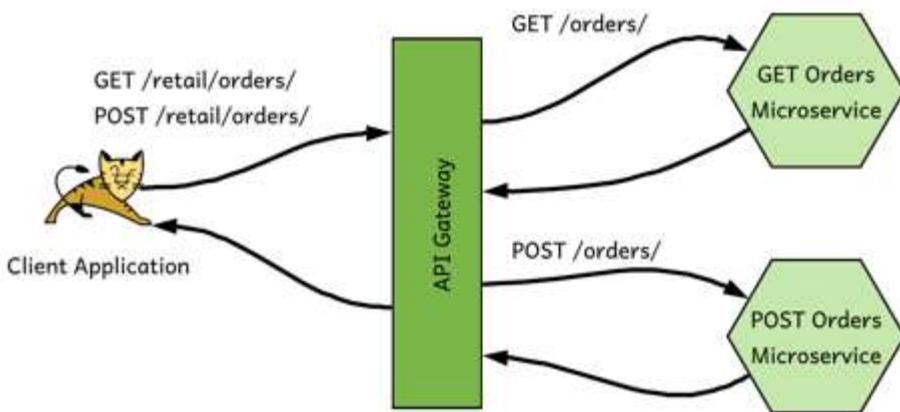


Figure 3.5 Multiple microservices are being exposed as a single API on the gateway. The client application only needs to worry about a single endpoint.

3.2 Security at the edge

In this section, we look at why OAuth 2.0 is the most appropriate protocol for securing your microservices at the edge. In a typical microservices deployment, we do not directly expose microservices to client applications. The API gateway, which is the entry point to the microservices deployment, selectively exposes microservices as APIs to the client applications. In most cases, these API gateways use OAuth 2.0 as the security protocol to secure the APIs they expose at the edge. If you are interested in understanding OAuth 2.0 and API security in detail, we would recommend you to have a look at the book, Advanced API Security: OAuth 2.0 and Beyond (Apress, 2019) by Prabath Siriwardena (a co-author of this book). OAuth 2 in Action (Manning Publications, 2017) by Justin Richer and Antonio Sanso is also a very good reference on OAuth 2.0.

3.2.1 Understanding the consumer landscape of your microservices

As discussed earlier in this chapter, the primary reason why organizations and enterprises adopt microservices is the agility that microservices provide for developing services. An organization wants to be agile to develop and deploy services as fast as possible. The pace is driven by the rise of demand in consumer applications. Today, people use mobile applications for most of their day-to-day activities, such as ordering pizza, grocery shopping, networking, interacting socially, and banking. These mobile applications consume services from various providers.

In an organization, both its internal and external (such as third-party applications) applications, could consume microservices. External applications could be mobile applications, web applications on the public Internet, applications running on devices or cars, and so on. For these types of applications to work, you need to expose your microservices over the public Internet over HTTPS. As a result, you cannot just rely on network level security policies to prevent access to these microservices. Therefore, you may always have to rely on an upper layer of security to control access. An upper layer of security here refers to the layers in the TCP/IP protocol stack (<https://www.w3.org/People/Frystyk/thesis/TcpIp.html>). You need to rely on security that's applied above the Network layer, such as Transport- or Application-layer protocols, such as TLS and HTTPS.

Applications running within the organization's computing infrastructure may consume both internal-facing and external-facing microservices. Internal-facing microservices may also be consumed by other microservices that are external-facing or internal-facing. As shown in figure 3.6, in the retail-store example, the microservice that's used for browsing the product catalog (the Products microservice) and the microservice that's used for taking orders (the Order Processing microservice) are external-facing microservices that are required by applications running outside the security perimeters of the organization. But the microservice that's used for updating the inventory—the Inventory microservice—need not be exposed outside the organization's security perimeters, because the inventory is updated only when an order is placed (via the Order Processing microservice) or when stocks are added to inventory through an internal application.

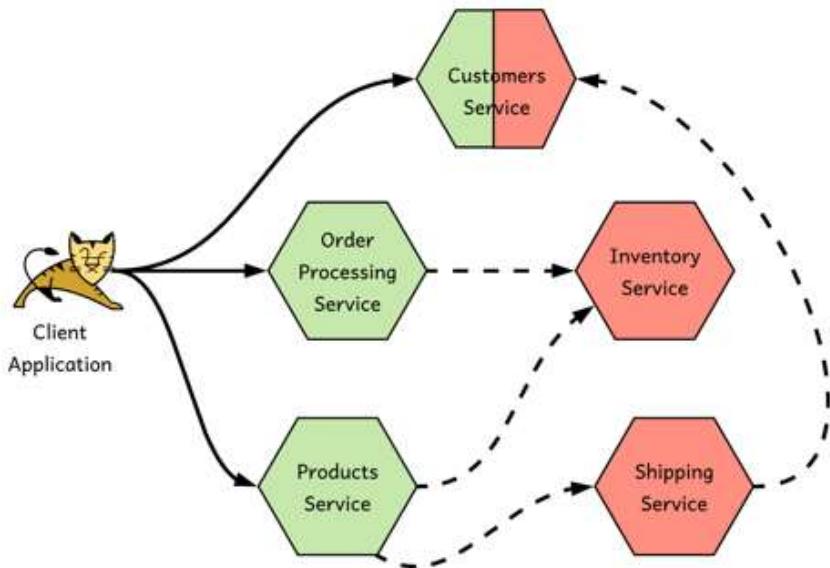


Figure 3.6 Internal microservices, external microservices, and hybrid microservices, each communicating with others to fulfill their functionality

3.2.2 Delegating access

A microservice is exposed to the outside world as an API. As for microservices, for APIs the audience is a system that acts on behalf of itself or on behalf of a human user or another system. It's unlikely (but not impossible) for human users to interact directly with APIs. This is where the access delegation is important and plays a key role in securing APIs. As we discussed briefly in chapter 2, multiple parties are involved in a typical flow to access a secured microservice (figure 3.7). Even though we didn't worry about APIs in our chapter 2 discussions for simplicity, the flow does not deviate a lot even if we introduce an API, in between the client application and the Order Processing microservice.

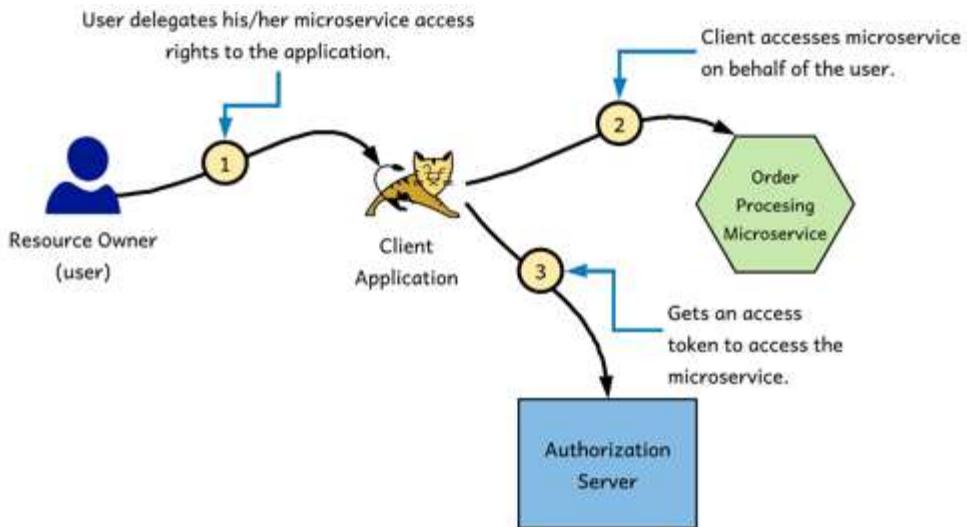


Figure 3.7 Multiple parties are involved in a typical flow to access a microservice, protected with OAuth.

A user (resource owner) should be allowed to perform only the actions on microservices that he or she is privileged to perform. The data that the user retrieves from the microservice or updates via the microservice should be only the data that he or she is entitled to receive or update. Although this level of privilege is checked against the user, the entity that accesses the microservice on behalf of the user is the client application the user uses. In other words, the actions that the user is entitled to perform on the microservices are executed by a client application. In effect, the user delegates his or her access rights to an application that accesses the resources on the microservices. As a result, the application has a responsibility to deal with the delegated rights appropriately. Therefore, the trustworthiness of the application is important. Especially when third-party applications are being used to access resources on your microservices, having a mechanism that allows you to control which actions the application can perform on your resources becomes important. Controlling the delegation of access to client applications is an essential factor in deciding on a mechanism to secure your microservices.

3.2.3 Why not basic authentication to secure APIs?

Basic authentication allows a user (or a system) with a valid username and password to access a microservice via an API. In fact, basic authentication (or basic auth) is a standard security protocol introduced with HTTP/1.0 RFC long time back. It allows you to pass the base64-encoded username and password, in the HTTP Authorization header, along with a request to an API. This model fails to meet access delegation requirements we discussed in section 3.2.2 in a microservices deployment, though, for a variety of reasons:

- *The username and password are static, long-living credentials.* If a user provides his or

her username and password to an application, the application needs to retain this information for that particular user session to access the microservices. The time during which this information needs to be retained could be as long as the application decides. None of us likes having to authenticate into an application again and again to perform operations. Therefore, if Basic authentication is used, the application has to retain this information for long durations of time. The longer this information is retained, the higher the chance of compromise. And because these credentials almost never change, a compromise of this information could have severe consequences.

- *No restrictions on what the application can do.* After an application gets access to the username and password of a user, it can do everything that user can do with the microservice. Not just accessing the microservice, the application can do anything with those credentials, even on other systems.

3.2.4 Why not Mutual TLS to secure APIs?

Mutual Transport Layer Security (mTLS) is a mechanism in which a client application verifies a server and the server verifies the client application by exchanging respective certificates and proving each one owns the corresponding private keys. In chapter 6, we discuss in detail about mTLS. For the moment, think of mTLS as being a mechanism for building two-way trust between client application and server using certificates.

mTLS solves one of the problems with basic authentication by having a lifetime for its certificates. The certificates used in mTLS are time-bound, and whenever a certificate expires, it's no longer considered to be valid. Therefore, even if a certificate and the corresponding private key are compromised, its vulnerability is limited by its lifetime. In some situations, however, certificates have lifetimes as long as years, so the value of mTLS over protocols such as Basic authentication is limited.

Just like in Basic authentication, mTLS too fails to meet access delegation requirements we discussed in section 3.2.2 in a microservices deployment. mTLS doesn't provide a mechanism to represent the end-user who uses the corresponding application. You can use mTLS to authenticate the client application using the microservice but it does not represent the end-user. If you want to pass the end-user information with mTLS, you would need to follow your own custom techniques, such as sending the username as a custom HTTP header; which is not quite recommended. Therefore, mTLS is mostly used to secure communication between a client application and a microservice or the communication between microservices. In other words, mTLS is mostly used to secure communications between systems.

3.2.5 Why OAuth 2.0?

To understand why OAuth 2.0 is the best security protocol for securing your microservices at the edge, you need to understand the problems related to microservices security. Security is all about granting controlled access to resources on a microservice. And if security is all about granting access to your resources, you need to figure out who wants access to your resources,

for what purpose, and for how long. You must properly understand the audience of your microservices through their characteristics and desires.

- **Who:** Ensure that only permitted entities are granted access to your resources.
- **What purpose:** Ensure that the permitted entities can perform only what they're allowed to perform on your resources.
- **How long:** Ensure that access is granted for only the desired period.

NOTE Before running the samples in this chapter, please make sure that you have downloaded and installed all the required software as mentioned in section 2.1.1 of chapter 2.

3.3 Setting up an API gateway with Zuul

In the first part of this chapter, we stated why an API gateway is an important component of a microservices deployment. In this section, you set up an API gateway for your Order Processing microservice, using Zuul (<https://github.com/Netflix/zuul/wiki>) as the gateway. Zuul is an open-source proxy server built by Netflix, acting as the entry point for all of the company's backend streaming applications.

3.3.1 Compiling and running the Order Processing microservice

To begin, download the chapter 3 samples from GitHub (<https://github.com/microservices-security-in-action/samples>) to your computer. The examples in this chapter use Java version 11, but still should work with Java 8+. Before running the examples in this chapter, make sure that you've stopped running the examples from other chapters or elsewhere. You could experience port conflicts if you attempt to start multiple microservices on the same port.

Once you downloaded all the samples from GitHub repository, you should see a directory named `sample01` inside the `chapter03` directory. This is the same sample used in chapter 2; we repeat it here for the benefit of those who skipped that chapter. Navigate to the `chapter03/sample01` directory from your command-line client application, and execute the following command to build the source code of the Order Processing microservice:

```
\> mvn clean install
```

If the build is successful, you should see a message on the terminal saying BUILD SUCCESS. If you see this message, you can start the microservice by executing the following command from the same location:

```
\> mvn spring-boot:run
```

If the service started successfully, you should see a log statement on the terminal that says Started OrderApplication in <X> seconds. If you see this message your Order Processing microservice is up and running. Now send a request to it, using curl, to make sure that it responds properly.

```
\> curl -d "{\"items\": [{\"itemCode\": \"IT0001\", \"quantity\": 3}, {\"itemCode\": \"IT0004\", \"quantity\": 1}], \"shippingAddress\": \"No 4, Castro Street, Mountain\"}
```

```
View, CA, USA\"} -H "Content-Type: application/json" http://localhost:8080/orders
```

Upon successful execution of this request, you should see a response message:

```
{"orderId":"7c3fb57b-3198-4cf3-9911-6dd157b93333","items":[{"itemCode":"IT0001","quantity":3},{"itemCode":"IT0004","quantity":1}],"shippingAddress":"No 4, Castro Street, Mountain View, CA, USA"}
```

The above request gets curl (a client application) to access your Order Processing microservice directly, as shown in figure 3.8. Your client application sent a request to the Order Processing microservice to place an order. As you saw in the response message, the ID of the particular order is 7c3fb57b-3198-4cf3-9911-6dd157b93333. Later, when you try to retrieve the same order by using the GET /orders/{id} resource, you should be able to get the details on the order you placed.

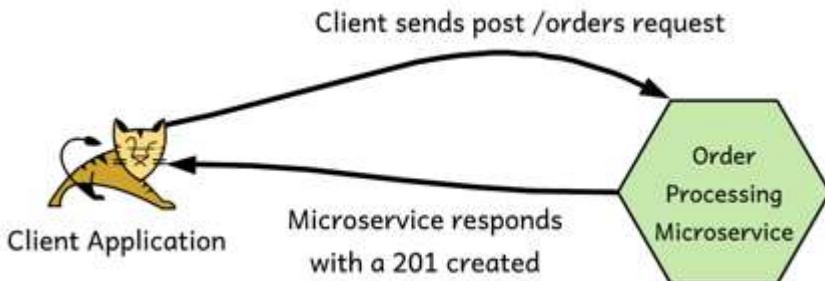


Figure 3.8 Client application sends a request directly to the Order Processing microservice and gets a response back with an order ID.

3.3.2 Compiling and running the Zuul proxy

The next step is compiling and running Zuul as a proxy to the Order Processing microservice. To build the Zuul proxy, navigate to the directory named chapter03/sample02 in your command-line client, and execute this command:

```
\> mvn clean install
```

You should see the BUILD SUCCESS message. Next, run the Zuul proxy by executing the following command from within the same directory:

```
\> mvn spring-boot:run
```

You should see the server-start-successful message. Now try to access your Order Processing microservice through the Zuul proxy. To do so, you'll be attempting to retrieve the details on the order you placed. Execute the following command from your terminal application (make sure to have the correct order ID from section 3.3.1):

```
\> curl http://localhost:9090/retail/orders/7c3fb57b-3198-4cf3-9911-6dd157b93333
```

If the request is successful, you should see a response like this:

```
{"orderId":"7c3fb57b-3198-4cf3-9911-  
6dd157b93333","items":[{"itemCode":"IT0001","quantity":3}, {"itemCode":"IT0004","quantit  
y":1}],"shippingAddress":"No 4, Castro Street, Mountain View, CA, USA"}
```

This response should contain the details of the order you created earlier. There are several important points to note in this request:

- As you may have noticed, the port to which you sent the request this time (9090) isn't the same as the port of the Order Processing microservice (8080) because you're sending the request to the Zuul proxy instead of the Order Processing microservice directly.

The next important thing to note is the URL. The request URL now starts with /retail, which is the base path in Zuul that you've configured to route requests to the Order Processing microservice. To see how routing is configured, open the application.properties file that resides in the sample02/src/main/resources directory by using a text editor. The following line you find there instructs the Zuul proxy to route requests received on /retail to the server running on http://localhost:8080.

```
zuul.routes.retail.url=http://localhost:8080
```

Figure 3.9 illustrates how Zuul does routing by dispatching a request it gets from the client application to the Order Processing microservice.

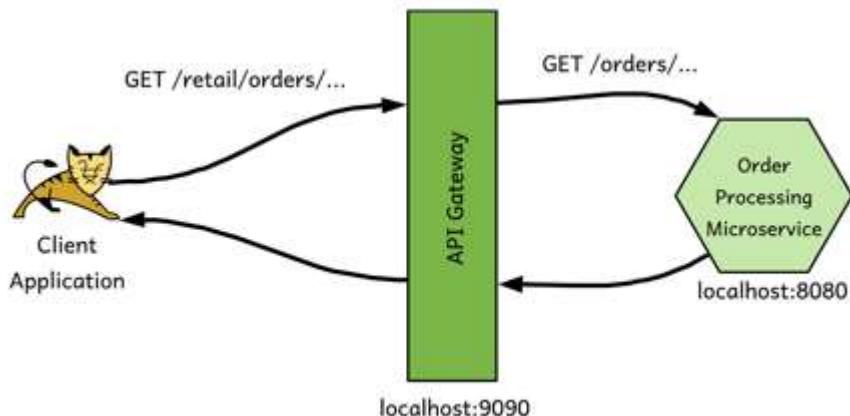


Figure 3.9 The Order Processing microservice is proxied via the Zuul gateway. All requests to the microservice need to go through the gateway.

3.3.3 Enabling OAuth 2.0-based security at the Zuul gateway

Now that you've successfully proxied your Order Processing microservice via the Zuul gateway, the next step is enforcing security on the Zuul gateway so that only authenticated clients are granted access to the microservice. First, you need an authorization server (see appendix D), which is capable of issuing access tokens to clients. In typical production deployment architecture, the authorization server is deployed inside the organization's network, and only the required endpoints are exposed externally. Usually, the API gateway is the only component that's allowed access from outside; everything else is restricted within the local area network of the organization. In the examples in this section, the /oauth2/token endpoint of the authorization server is exposed through the Zuul gateway so that clients can obtain access tokens from the authorization server. Figure 3.10 illustrates this deployment architecture.

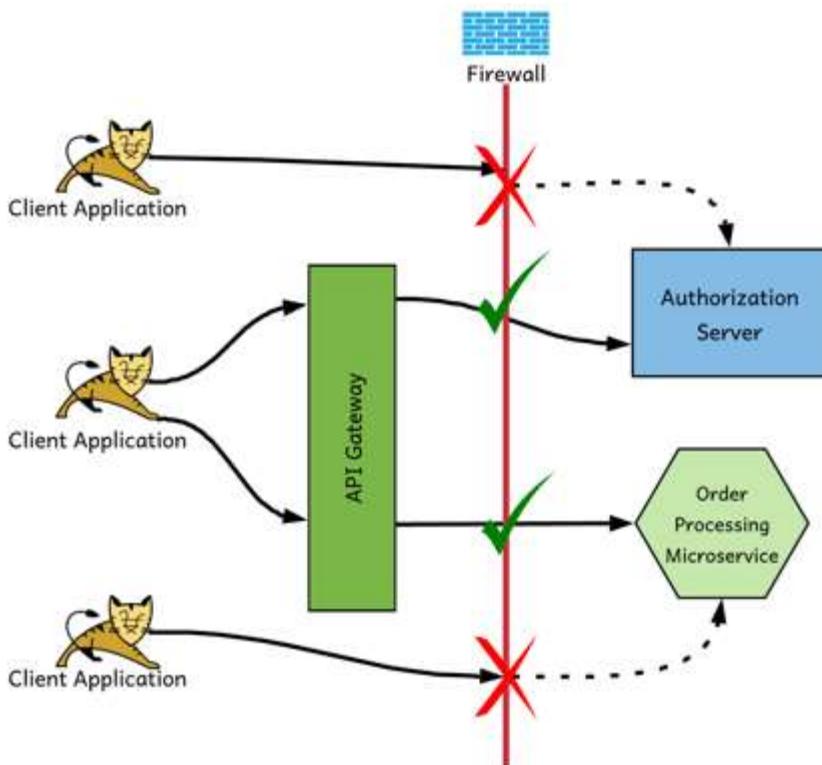


Figure 3.10 The firewall guarantees that access to the authorization server and microservice can happen only via the API gateway.

To build the authorization server, navigate to the `chapter03/sample03` directory from your command-line client, and execute the following command:

```
\> mvn clean install
```

When the authorization server is built, you can start it by using the following command:

```
\> mvn spring-boot:run
```

When the authorization server starts successfully, you can request tokens from it via the Zuul gateway. You may have noticed a line in the sample03/src/main/resources/application.properties file that routes requests received on the /token endpoint of Zuul to the authorization server:

```
zuul.routes.token.url=http://localhost:8085
```

The following curl command gives you an access token from the authorization server via the Zuul gateway:

```
\> curl -u application1:application1secret -H "Content-Type: application/x-www-form-urlencoded" -d "grant_type=client_credentials" http://localhost:9090/token/oauth/token
```

You should receive the access token in a response that looks like this:

```
{"access_token": "47190af1-624c-48a6-988d-f4319d36b7f4", "token_type": "bearer", "expires_in": 3599}
```

ENFORCING VALIDATION OF TOKENS AT THE ZUUL GATEWAY

Once the client application gets an access token from the token endpoint of the authorization server, next, it accesses the Order Processing microservice via the Zuul gateway, with this access token. The purpose of exposing the Order Processing microservice via the Zuul gateway is to make the gateway enforce all security-related policies while the Order Processing microservice focuses only on the business logic it executes. This situation is in line with the principles of microservices, which state that a microservice should focus on doing only one thing. Now you need to make sure that the Zuul gateway allows requests to the Order Processing microservice only if the requesting client bears a valid access token.

Because this gateway didn't issue the access token to the requesting client, it doesn't know how to validate this access token. Therefore, it needs to talk to the authorization server that issued the access token to check for its validity. This process is known as *token introspection* (<https://tools.ietf.org/html/rfc7662>). The request flow from client to microservice is shown in figure 3.11.

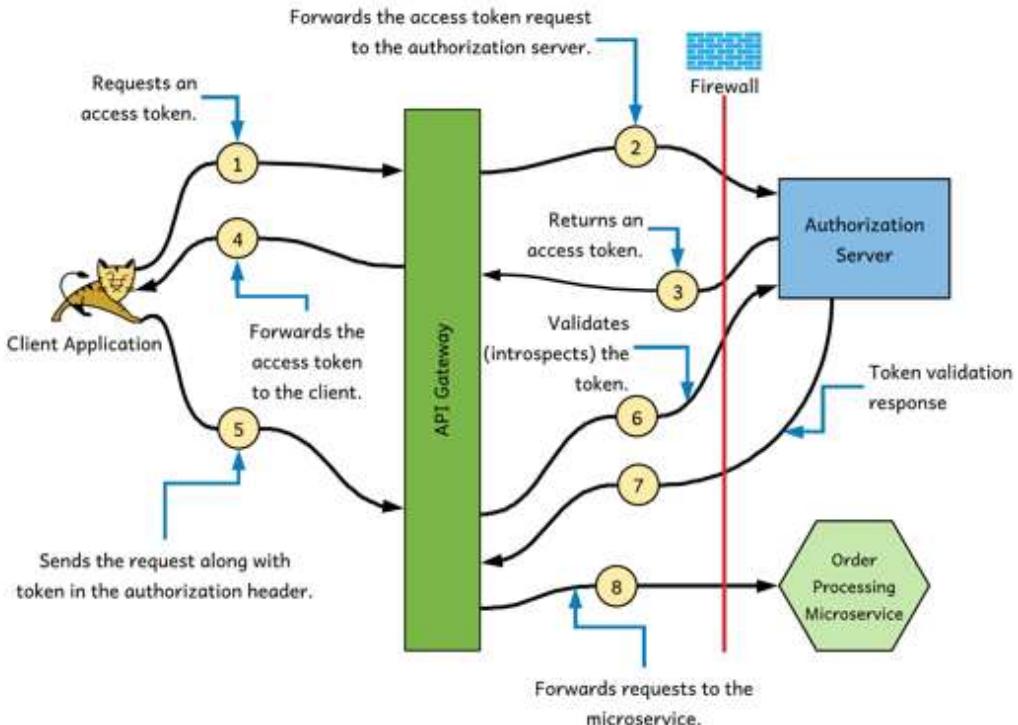


Figure 3.11 Message exchanges from the point where the client application gets a token to the point where it accesses the microservice.

As shown in figure 3.11, the client application sends an OAuth2.0 access token as a header to the Zuul gateway on the path on which the Order Processing microservice is exposed (/retail/orders). The gateway extracts the token from the header and introspects it through the authorization server. The authorization server responds with a valid or invalid status message; if the status is valid, the gateway allows the request to be passed to the Order Processing microservice. To do this in Zuul, you use a request filter, which intercepts requests and performs various operations on them. A filter can be one of four types:

- *Prerequest filter*—A filter that's executes before the request is routed to the target service
- *Route filter*—A filter that can handle the routing of a message
- *Post-request filter*—A filter that's executes after the request has been routed to the target service
- *Error filter*—A filter that's executes if an error occurs in the routing of a request

In this case, because you need to engage the validation before routing the request to the target service, you use a prerequest filter. You can find the source code of this filter in the following class

```
sample04/src/main/java/com/manning/mss/ch03/sample04/filters/OAuthFilter.java
```

If you inspect the contents of this Java class, you notice a method named `filterType`. This method returns a string `as pre`. This string tells the Zuul runtime that it's a prerequest filter that needs to be engaged before the request is routed to the target service. The `run` method of this class contains the logic related to introspecting the token through the authorization server. If you use that method, you'll notice that a few validations are performed on the Zuul gateway itself to check whether the client is sending the access token in an HTTP header named `Authorization` and whether the header is received in the correct format. When all format checks are done, the gateway talks to the authorization server to check whether the token is valid. If the authorization server responds with an HTTP response status code of 200, then it is a valid token:

```
int responseCode = connection.getResponseCode();

//If the authorization server doesn't respond with a 200.
if (responseCode != 200) {
    log.error("Response code from authz server is " + responseCode);
    handleError(requestContext);
}
```

If the server doesn't respond with 200, then the authentication has failed. The authentication could have failed for many reasons. The token may have been incorrect, the token may have expired, the authorization server may have been unavailable, and so on. At this point, you're not concerned about the reason for the failure. Unless the authorization server responds with 200, consider the authentication has failed.

NOTE The examples above cover a fundamental mechanism of how you can apply OAuth 2.0 based security on your microservices through an API gateway. You may not understand the source code in the samples in full, what is important is that you understand the pattern we are applying to secure your microservices.

OAUTH2.0 TOKEN INTROSPECTION PROFILE

We talked briefly about OAuth 2.0 token introspection (<https://tools.ietf.org/html/rfc7662>) in the preceding section. There the API gateway makes a request to the authorization server to validate a given token. Following is what the token introspection request looks like:

```
POST /introspection
Content-Type: application/x-www-form-urlencoded
Authorization: Basic YXBwbGljYXRpb24xOmFwcGxpY2F0aw9uMXNlY3JldA==

token=626e34e6-002d-4d53-9656-9e06a5e7e0dc&
token_type_hint=access_token&
resource_id=http://resource.domain.com
```

As you can see, the introspection endpoint of the authorization server is protected with basic authentication. The introspection request is sent to the authorization server as a typical form submit with the content type `application/x-www-form-urlencoded`. The `token` field contains the value of the token that you want to check for validity. The `token_type` field indicates to the authorization server whether this token is an `access_token` or `refresh_token`. The `resource_id` field indicates the resource on which you want to perform the introspection. When the authorization server completes the introspection, it responds with a payload similar to this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "active": true,
    "client_id": "asdh7j-uiwe83-8a73ki",
    "scope": "read write",
    "sub": "nuwan",
    "aud": "http://resource.domain.com"
}
```

The `active` field indicates that the token is active (not expired). The `client_id` is the identifier of the application for which the token was issued. The `scope` field includes the scopes bound to the token. The `sub` field contains the identifier (username) of the entity that consented to the token, the token owner. The `aud` field indicates a list of identifiers of the entities that are considered to be valid receivers/users of this access token.

Using the information in the introspection response, the gateway can allow or refuse access to the resources. It can also perform fine-grained access control (authorization) based on the scopes and also get to know what is the client application (from the `client_id`) that's requesting access to its resources.

SELF-VALIDATION OF TOKENS WITHOUT INTEGRATING WITH AN AUTHORIZATION SERVER

The key benefit of using microservices for your architectures is the agility it provides developers in terms of developing and managing software systems. The fact that microservices can operate by themselves without affecting other components in the system/architecture is one important factor that gives developers this agility. But the gateway component relies heavily on the authorization server to enable access to your microservices, so the gateway component is coupled with another entity. Although you may achieve agility on your microservices layer, the fronting layer (which is the API gateway) can't be managed with the same level of agility due to its reliance on the authorization server for token validations.

If you're a microservice developer who wants to put an API gateway in front of your microservice, this architecture doesn't give you the necessary flexibility; you have to come to an agreement with the administrator of the authorization server to get a set of credentials to access its introspection endpoint. If you're running this system in production and want to scale up your microservice and the API gateway to deal with a high number of requests, the

performance of your authorization server will be affected, because the gateway will call it each time it wants to validate a token. An authorization server, unlike other services in an organization, usually is managed by a separate group of people who have special privileges due to the server's sensitivity. Therefore, you can't expect the same level of dynamic scaling capabilities on your authorization server to meet the demands of the API gateway.

The way to deal with this problem is to find a mechanism that enables the gateway to validate tokens by itself without the assistance of an authorization server. To see how, look at what an authorization server does when someone asks it to validate a token through an introspection call:

1. It checks to see whether that particular token exists in its token store (database). This step verifies that the token was issued by itself and that the server knows details about that token.
2. It checks whether the token is still valid (token state, expiry, and so on).
3. Based on the outcome of these checks, it responds to the requester with the information discussed under the OAuth 2.0 Token Introspection section.

If the access token received on the API gateway, instead of being an opaque meaningless string, contains all the information you need, including expiry details, scopes, client id, user and so on, the gateway could validate the token by itself. But anybody could create this string and send it along to the gateway. The gateway wouldn't know whether a trusted authorization server issued the token.

JSON Web Tokens (JWTs) are designed to solve this problem (see appendix H for the details). A JSON Web Signature (JWS) is a signed JWT by the authorization server. By verifying the signature of the JWS, the gateway knows that this token was issued by a trusted party and that it can trust the information contained in the body. The standard claims described in the JWT specification (<https://tools.ietf.org/html/rfc7519>) don't have placeholders for all the information that's included in the introspection profile, such as the `client_id` and `scope`. The authorization server can add whatever information is missing to the JWT as custom claims. We discuss about self-contained access tokens in detail, in appendix D.

PITFALLS OF SELF-VALIDATING TOKENS AND HOW TO AVOID THEM

The self-validating token mechanism discussed in the preceding section comes at a cost, with pitfalls that you have to be mindful of. In the event that one of these tokens is prematurely revoked, the API gateway wouldn't know that the token has been revoked, because the revocation happens at the authorization server end, and the gateway no longer communicates with the authorization server for the validation of tokens.

One way to solve this problem is to issue short-lived JWTs (tokens) to client applications to minimize the period during which a revoked token will be considered to be valid on the API gateway. In practice, however, applications with longer user sessions have to keep refreshing their access tokens, when the tokens carry a shorter expiration.

Another solution is for the authorization server to inform the API gateway whenever a token has been revoked. The gateway and authorization server can maintain this communication channel via a pub-sub (<https://cloud.google.com/pubsub/docs/overview>) mechanism. This way, whenever a token is revoked, the gateway receives a message from the authorization server through a message broker. Then the gateway can maintain a list of revoked tokens until their expiry and before validating a given token check if it exists in the “revoked tokens” list. Revocations are rare, however. Figure 3.12 illustrates the revocation flow.

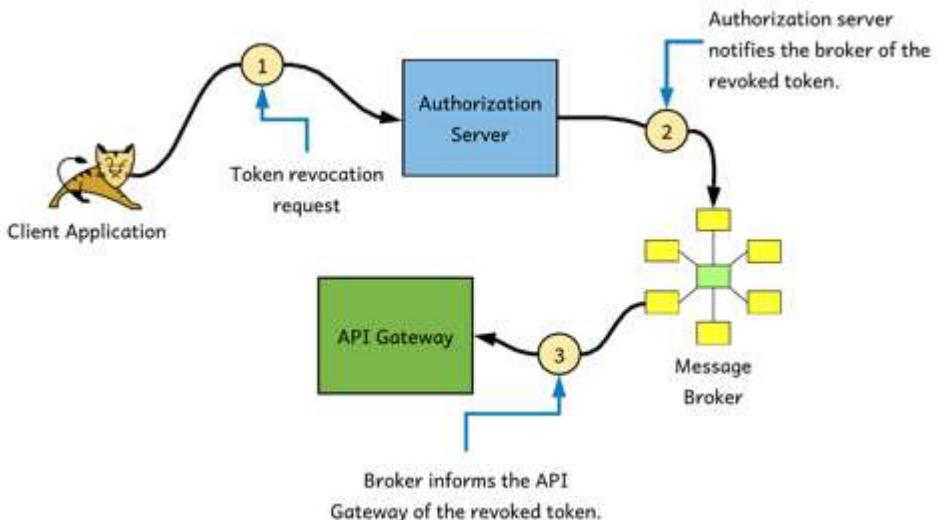


Figure 3.12 Executing the token revocation flow via a pub-sub mechanism. Upon a token revocation, the authorization server notifies the message broker, hence the API gateway.

Another problem with the self-contained access token is that the certificate used to verify token signature might have expired. When this happens, the gateway can no longer verify the signature of an incoming JWT (as access token). To solve this problem, you need to make sure that whenever a certificate is renewed, to deploy the new certificate on the gateway. Also, if the certificate, which is used by the authorization server, is revoked, then that decision too must be communicated to the API gateway.

3.4 Securing communication between Zuul and the microservice

So far, you've used the API Gateway pattern to secure access to your microservice. This pattern ensures that no one who doesn't have valid credentials (a token) gets access to your microservice through the API gateway. But you also have to consider what happens if someone accesses the microservice directly, bypassing the API gateway layer. In this section, we discuss how to secure access to your microservice in such case.

3.4.1 Preventing access through the firewall

First and foremost, you need to make sure that your microservice isn't directly exposed to external clients, so you need to make sure that it sits behind your organization's firewall. That way, no external clients get access to your microservices unless they come in via the API gateway (see figure 3.13).

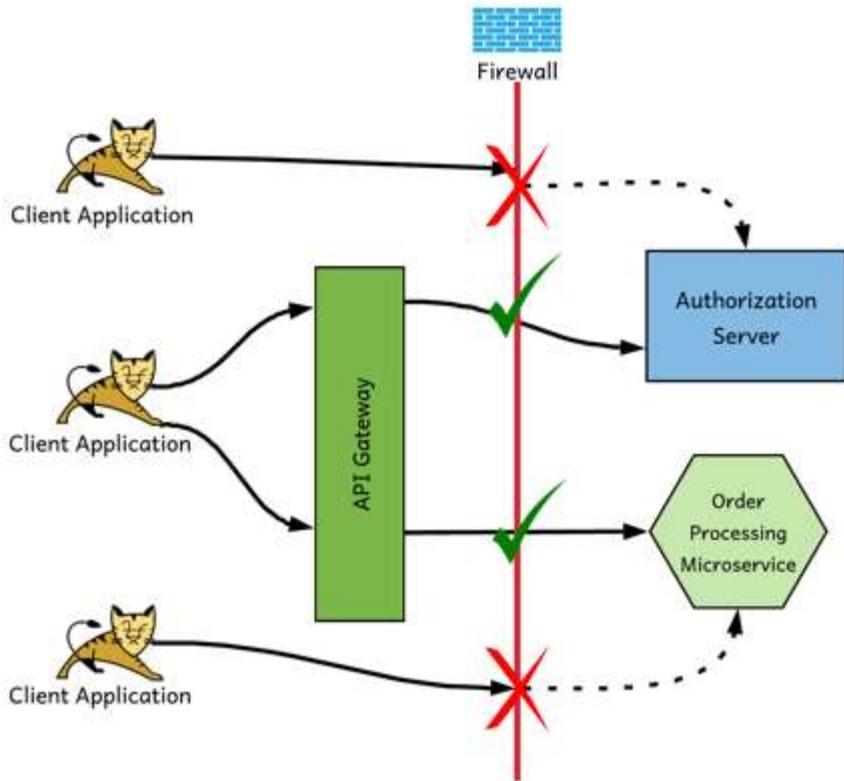


Figure 3.13 Direct access to the authorization server and microservice is prevented by the firewall, allowing only the API gateway to access them.

Although the API Gateway pattern ensures that no one outside the organization gets access to the microservice, a risk still exists that unintended internal clients may gain access to the microservice. The following section discusses how to prevent this situation.

3.4.2 Securing the communication between the API gateway and microservices by using mutual TLS

To make sure that your microservice is secure from internal clients and accessible only via the API gateway, you need to build a mechanism in which the microservice rejects any requests coming from clients other than the API gateway. The standard way is to enable mutual TLS (mTLS) between the API gateway and the microservice. When you use mTLS, you get the microservice to verify the client that talks to it. If it trusts the API gateway, then the API gateway can route requests to the microservice.

Under microservices principles, it's not a good idea for a microservice to be performing many operations. In fact, it's said that the microservice should focus on doing only one thing: executing the business logic that it's supposed to execute. You may think that you're burdening the microservice with extra responsibilities by expecting it to verify the client through mTLS. But mTLS verification happens at the transport layer of the microservice and doesn't propagate up to the application layer. Microservices developers don't have to write any application logic to handle the client verification, which is done by the underlying transport-layer implementation. Therefore, mTLS verification doesn't affect the agility of developing the microservice itself and can be used safely without violating microservices principles.

If you still worried about doing certificate validation as part of your microservice, in chapter 12, we discuss an approach to avoid that using the Service Mesh pattern. With the Service Mesh pattern, a proxy, which runs along with your microservice (each microservice has its own proxy) intercepts all the requests and does the security validation. The certificate validation can be part of that as well.

In chapter 6, we cover using client certificates to secure service-to-service communication. This scenario is essentially the one in which the API gateway becomes the source service and the microservice becomes the target service. We'll go into the details on setting up the certificates and building trust between clients and services in chapter 6. Figure 3.14 illustrates how both internal and external client applications are permitted to access a microservice only via the API gateway.

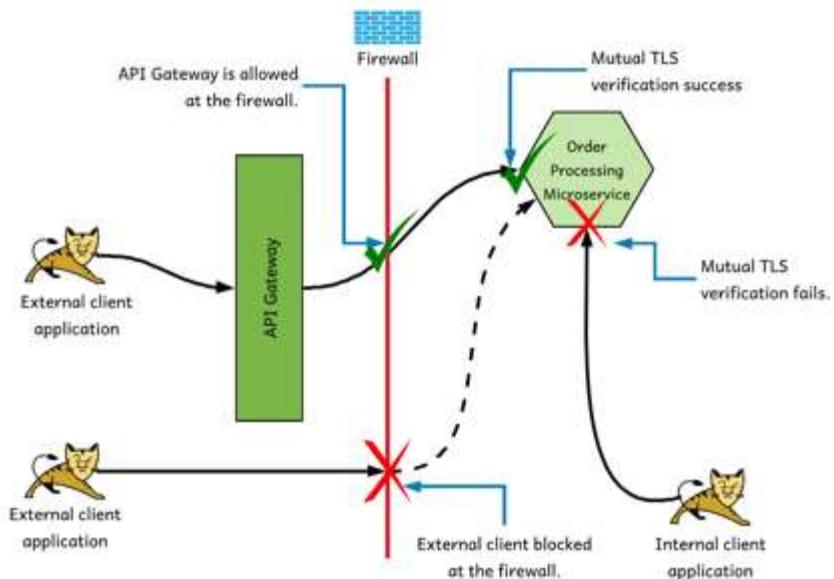


Figure 3.14 Enabling mutual TLS at the microservice to prevent access by unintended parties. All the requests outside the firewall must be routed via the API gateway.

3.5 Summary

- The API Gateway pattern is used to expose microservices to client applications as APIs.
- The API gateway helps to expose microservices of different flavors using a consistent and easy to understand interface to the consumers of these microservices.
- We do not have to expose all microservices through the API gateway. Some microservices are consumed internally only, in which case they will not be exposed through the gateway to the outside world.
- Protocols such as basic authentication and mutual TLS are not sufficient to secure APIs and microservices are exposed to the outside world via APIs.
- OAuth 2.0 is the de facto standard for securing APIs and microservices at the edge.
- OAuth 2.0 is an extensible authorization framework, which has a wide array of grant types. Each grant type defines the protocol of how a client application would obtain an access token to access an API/microservice.
- We need to choose the right OAuth 2.0 grant type for our client applications based on their security characteristics and trustworthiness.
- An access token can be a reference token or a self-contained token (JWT). If it is a reference token, then the gateway has to talk to the issuer (or the authorization server) all the time to validate it. For self-contained tokens, the gateway can do the validation by verifying the token signature.

- A self-contained access token has its own pitfalls. It is recommended to have short-lived JWTs for self-contained access tokens.
- The communication between the gateway and the microservice can be protected either with firewall rules or mutual TLS – or a combination of both.

4

Accessing a secured microservice via a single-page application

This chapter covers

- What is a single-page application (SPA), its benefits and drawbacks
- How to build a SPA using Angular and Spring Boot to talk to a secured microservice
- How to overcome Cross-Origin Resource Sharing (CORS) related issues
- How to login to a SPA with OpenID Connect

In chapter 2, we discussed how to secure a microservice with OAuth 2.0 and directly invoked it with a curl client. Chapter 3 further improved it by deploying the microservice behind an API gateway. The API gateway took over the OAuth 2.0 token validation responsibility from the microservice and the communication between the API gateway and the microservice was secured with mutual Transport Layer Security (mTLS). The API gateway introduced a layer of abstraction, between the client applications and the microservices. All the communications with microservices had to go through the API gateway. In this chapter, we discuss in detail how to build a single page application (pronounced as spä) to invoke microservices, via an API gateway. In case you are in doubt, why we talk about building a SPA in a microservices security book, the reason is; we believe in completing an end-to-end architecture with a microservices deployment, from data to screen. And SPAs are the mostly used client application type. Understanding the constructs of a SPA is important in building an end-to-end security design. If you are new to SPA architecture, we recommend you go through the appendix E first. It helps you understand what a SPA is and what benefits it offers.

4.1 Running a single-page application with Angular

Suppose that you've chosen to adopt SPA architecture to build your retail-store web application. In this section, you build a SPA by using Angular (<https://angular.io/>). Angular is a framework that helps you build web pages with dynamic HTML content. You'll implement a simple SPA so that you get first-hand experience with its characteristics and behaviors.

4.1.1 Building and running an Angular application from the source code

To start, you need the samples from the GitHub repo at <https://github.com/microservices-security-in-action/samples>. The samples related to this chapter are in the directory `chapter04`. Also make sure that none of the processes you may have started when trying samples from other chapters are running. Shut them down before you try any of the samples in this chapter. Also, before running the samples in this chapter, please make sure that you have downloaded and installed all the required software as mentioned in section 2.1.1 of chapter 2.

After you've set up the all dependencies and downloaded the relevant code on to your working environment, open a command-line client, and navigate to the `chapter04/sample01` directory. Within that directory, execute the following command to build `sample01`:

```
\> mvn clean install
```

If the build is successful, you should see a message on the terminal saying `BUILD SUCCESS`. If you see this message, start your Angular application by executing this command:

```
\> mvn spring-boot:run
```

If your application started successfully, you should see a message like this on the terminal:

```
Started UiApplication in <X> seconds
```

If you see this message, you can access the application. First, try to invoke the application with curl. Open a new window in your command-line client application, and execute the following command:

```
\> curl localhost:8080 -v
```

You should see an error message on the terminal.

```
{"timestamp":1539253933332,"status":401,"error":"Unauthorized","message":"Full authentication is required to access this resource","path":"/"}
```

You should also see following response headers from the server.

```
< HTTP/1.1 401
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 1; mode=block
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

```
< WWW-Authenticate: Basic realm="Spring"
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Thu, 11 Oct 2018 10:32:13 GMT
```

The response code 401 means that the server has rejected serving your request for content due to an authentication failure. The header `WWW-Authenticate` indicates that the server expects you to login by using basic authentication.

4.1.2 Behind the scenes of a single-page application

Try to access the same URL pointing to the SPA, which we used in section 4.1.1 using a browser. Use a private browsing session so that you can observe the behavior we explain in the following paragraphs in this section.

Open your web browser (in a private / incognito window) and then open the browser's developer tools so you can inspect what happens behind the scenes when you ask the server for content. (In Firefox, the developer tools usually open when you press the F12 key on your keyboard.) When the developer tools are open, go to the **Network** tab of the developer-tools window and click the **Persist Logs** button so you can observe the requests and responses exchanged by the web browser and the web server. Then type the following URL in the address bar of your browser, and press **Enter**.

```
http://localhost:8080/
```

The browser prompts you to enter a username–password pair. As you saw when you made the curl request, the server responds to the browser's request with 401 HTTP status code indicating that it requires basic authentication credentials. The browser understands this response and prompts the user to enter his or her basic authentication credentials. Enter `user` as the username and `password` as the password. When you enter the credentials, you should see some content in the browser window, including a message that says Welcome to your first single-page application, `user!` The `user` part of the message is the username you used to log in to the application. Figure 4.1 illustrates this message exchange.

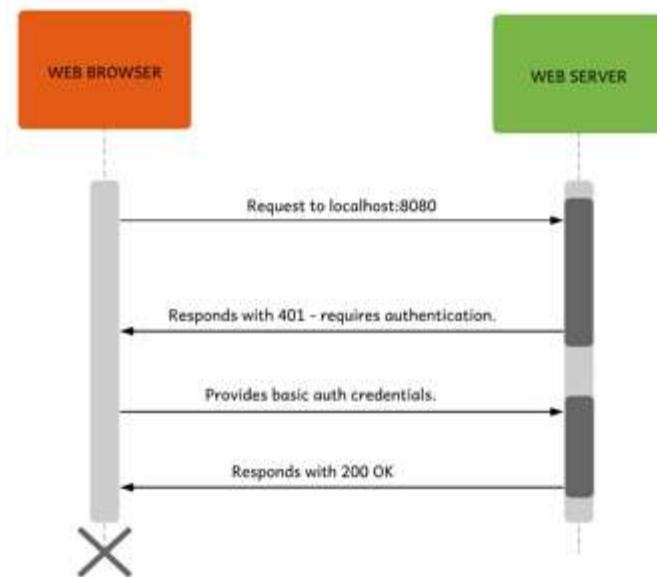


Figure 4.1 When a request hits an endpoint secured with basic authentication with no credentials, the web server responds back with the HTTP status code 401.

Now, let's take a look at the source code that generates this content. As we mentioned earlier, you're using an Angular web application hosted on Spring Boot. In discussing the benefits of an SPA in appendix E, we mentioned that because SPAs have simple design and content, they can be deployed on any simple HTTP hosting service. You use Spring Boot to host your SPA. This web application, like any standard web application, has an index file named `index.html`. The file is located at `sample01/src/main/index.html`. You can open this file with any text editor or an IDE. The interesting feature of this HTML file is the content of the `<body>` element:

```
<body>
  <app></app>
</body>
```

As you may notice, the `<app>` element isn't a standard DOCTYPE (https://www.w3schools.com/tags/ref_html_dtd.asp) in HTML. This particular `index.html` is loaded into the browser after you authenticate to the server; then the browser replaces the content of this element that's loaded into it. The JavaScript code running on the browser performs this action, which is typical behavior of SPAs. Many applications are designed in such a way that `index.html` is the only static HTML file loaded into the browser. User actions (clicks) result in its contents being dynamically updated by the JavaScript running on the browser

itself. Note that although some applications follow SPA architecture and principles, they could have more than one static HTML file loaded for different actions.

The Network tab of the browser's developer tools shows the request and response exchanges between the browser and server. First, you should see an HTTP request that goes to the root (/) of the application. The server responds with a 401 HTTP status code to this request. This response prompts the browser to open the window that asks for the user's username and password. After this request, you should see a collection of requests for various JavaScript files. To all these requests, you should see the server responding with the requested files and 200 OK (HTTP status code). The final request to the /resource path of the server fetches the actual (dynamic) content to be displayed on the web page. If you observe the content of that response message in the browser's developer tools, you should see this JSON payload:

```
{"id":"7209c5b7-19fc-43a2-93af-a2280b4004d5","user":"user","content":"Hello World"}
```

The message you saw in your browser was constructed from the data contained in this JSON string. When the browser made a request to the /resource path to the server running on localhost:8080, it executed a method on your Spring Boot application. You can find that method in the file located at sample01/src/main/java/demo/UiApplication.java (see listing 4.1).

Listing 4.1. The content of the UiApplication.java file

```
@RequestMapping("/resource")
public Map<String, Object> home() {

    Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
    String user = authentication.getName();

    Map<String, Object> model = new HashMap<String, Object>();
    model.put("user", user);
    model.put("id", UUID.randomUUID().toString());
    model.put("content", "Hello World");
    return model;
}
```

The method annotation @RequestMapping("/resource") instructs the Spring Boot runtime to execute this method when a request is received on the /resource path. If you go through the rest of the method contents, you can figure out how it constructs the content of the JSON you saw earlier.

Another file that can help you understand the dynamic HTML content is the file located at sample01/src/app/app.component.html. This file contains the HTML that will eventually replace the content of the <body> section of index.html (see listing 4.2).

Listing 4.2. The content of the app.component.html file

```
<div style="text-align:center" class="container">
  <h1>
```

```
Welcome to your first Single Page Application, {{greeting.user}}!
</h1>
<div class="container">
  <p>Id: <span>{{greeting.id}}</span></p>
  <p>Message: <span>{{greeting.content}}</span></p>
</div>
</div>
```

Notice the placeholders such as `{greeting.user}`, which will be replaced by the JSON content sent by the `/resource` endpoint. The code inside the file `sample01/src/app/app.component.ts` is a Type Script file that, upon loading, initiates the request to the `/resource` endpoint, as the following piece of code shows.

```
constructor(private http: HttpClient) {
  http.get('resource').subscribe(data => this.greeting = data);
}
```

Notice the HTTP GET request to the `/resource` endpoint of the web server. The response of the request is assigned to a variable named `greeting`.

In this particular example, your web server—the one that hosted the UI application and the data endpoint (`/resource`) are both running on the same host (`localhost:8080`). In most real-world scenarios this won't be the same because the data endpoints, also referred as APIs in many cases, would be on different hosts. We look at a similar example in the section 4.2.

4.2 Introducing an API gateway, and setting up cross-origin resource sharing (CORS)

In the retail-store example, suppose that different teams in your organization are developing the microservices that power the store functionality. These teams could be hosting their microservices in different domains (such as `orders.retailstore.com` and `products.retailstore.com`). Earlier in this chapter under section 4.1, you built a SPA with the frontend and backend both hosted on the same host, similar to keeping your retail-store SPA and microservices in the same domain. Because this design isn't popular in practice, you'll evolve it to use a resource server (data APIs) hosted on a different domain or host. Figure 4.2 shows the components in action.

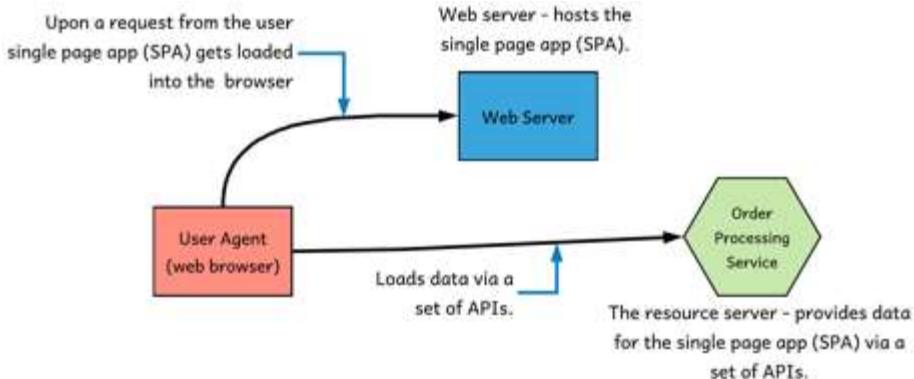


Figure 4.2 Web server hosts the SPA where the static content is loaded and it uses data APIs exposed by the resource server, which runs on a different host to build the content dynamically.

4.2.1 Running the sample

You can find the source code of this sample in the chapter04/sample02 directory. This directory contains two subdirectories: one for the UI application (web application) and the other for the resource server. Execute the steps in this section to run the sample. If you're running any samples from earlier sections, please shut them down before attempting the samples in this section. Use your command-line tool to navigate to the sample02 directory, and execute the following command:

```
\> mvn clean install
```

When the build is successful, use the command-line tool to navigate to the sample02/resource directory, and execute this command:

```
\> mvn spring-boot:run
```

This command should start the resource server. To see what resources it provides, execute the following curl command:

```
\> curl http://localhost:9000/orders
```

This command results in a set of orders being received as a JSON payload (see listing 4.3).

Listing 4.3. The JSON payload, which carries a set of orders

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {"orderId": "00001",
   "shippingAddress": "No 5, Castro Street, Mountain View, CA, USA.",
   "items": [
     {"itemCode": "IT001", "quantity": 3},
     {"itemCode": "IT002", "quantity": 5},
```

```
{
  "itemCode": "IT003", "quantity": 4}]],
  {"orderId": "00002",
  "shippingAddress": "No 20, 2nd Street, San Jose, CA, USA.",
  "items": [
    {"itemCode": "IT001", "quantity": 7},
    {"itemCode": "IT003", "quantity": 1}]}
]
```

This response (see listing 4.3) is from the resource server that runs on port 9000 of your host machine. Next, use the command-line client to navigate to the `sample02/ui` directory, and execute this command:

```
\> mvn spring-boot:run
```

When the process started successfully, open a private browsing session in your web browser and then go to the following URL:

```
http://localhost:8080
```

You see the message **Login to see your orders**. Click the **Login** button. At this point, open your web browser's developer tools; on the **Network** tab, select the options to preserve/persist logs and disable the cache. On the screen that appears, enter `user` as the username and `password` as the password, and submit. You should see an HTML table containing the details of the orders you observed through the curl command.

Here, you authenticated to the web application running on host `localhost:8080`; then your SPA/browser made a request to the resource server running on host `localhost:9000` to fetch details of some orders. The JavaScript running on the SPA/browser dynamically rendered a HTML table consisting of the information received from the resource server in JSON format. If you observe the network requests on the **Network** tab, you see an HTTP POST request to `http://localhost:8080/login`, which completed successfully with a 200 OK response. Next, you see an HTTP OPTIONS request to `http://localhost:9000/orders`. You didn't write code asking the application to make this request; the web browser automatically due to its same-origin policy, which we discuss in section 4.2.2, initiated it.

We just looked at how we can separate out the web application from the microservice that is actually executing the application logic. In the process of doing so we noticed an unusual request being sent from the web browser, which we haven't explicitly asked the web application to perform. In the section 4.2.2 we discuss in depth about this request and the need for it to be present.

4.2.2 The same-origin policy

The **same-origin policy** (https://en.wikipedia.org/wiki/Same-origin_policy) is a concept in web browsers introduced to ensure that scripts running on a particular web page can make requests only to services running on the same origin. An origin of a given URL consists of the URI scheme, hostname, and port. Given the URL `http://localhost:8080/login`, following are the sections that compose the origin.

- `http`—The URL scheme

- localhost—The hostname/IP-address
- 8080—The port

The sections after the port aren't considered to be part of the origin; therefore, /login isn't considered to be part of the origin. The same-origin policy exists to prevent a malicious script from one website accessing data from other websites unintentionally. The same-origin policy applies only to data access, not to CSS, images, and scripts, so you could write web pages that consist of links to CSS, images, and scripts of other origins. Figure 4.3 illustrates this scenario.

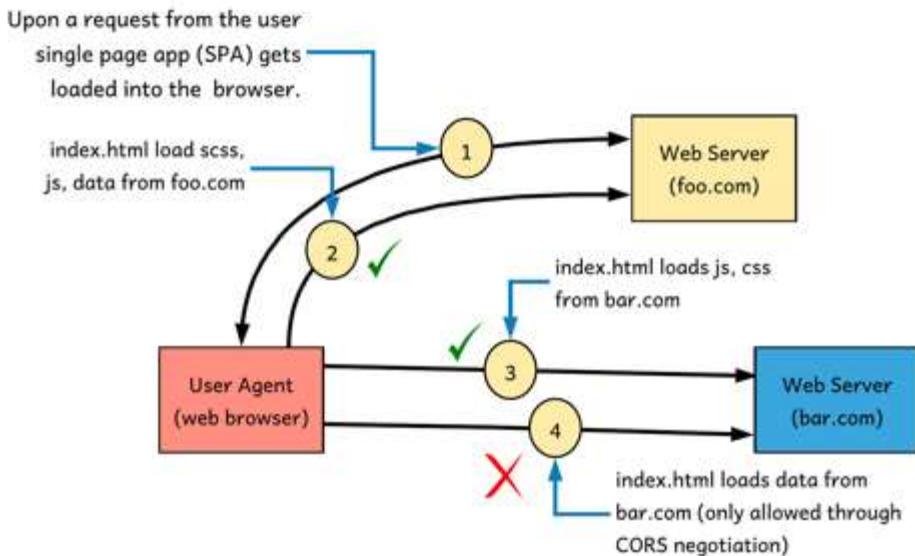


Figure 4.3 How web browsers apply the same-origin policy. The same-origin policy ensures that scripts running on a particular web page can make requests only to services running on the same origin.

Following is what happens in each step shown in figure 4.3.

- *Step 1*—The browser loads an HTML file (**index.html**) from the domain **foo.com**. This request is successful.
- *Step 2*—The **index.html** file loaded into the browser makes a request to the same domain (**foo.com**) to load some CSS and JavaScript files; it also loads data (makes an HTTP request) from the domain **foo.com**. All requests are successful because everything is from the same domain as the web page itself.
- *Step 3*—The **index.html** file loaded into the browser makes a request to a domain named **bar.com** to load some CSS and JavaScript files. This request, although made to a different domain (**bar.com**) from a web page loaded from another domain (**foo.com**) is successful because it's loading only CSS and JavaScript.
- *Step 4*—The **index.html** file loaded into the browser loads data (makes an HTTP request) from the domain **bar.com**. This request is blocked by a red 'X' because it's attempting to load data from a different origin without proper CORS negotiation.

from an endpoint on domain bar.com. This request fails because by default, the browser doesn't allow web pages in one domain (foo.com) to make HTTP requests to web pages in other domains (bar.com) unless the request is for CSS, JavaScript, or images.

What is the danger of not having a same-origin policy?

Suppose that you're logged into Gmail in your web browser. As you may know, Gmail uses cookies in your browser to maintain data related to your browser session. Someone sends you a link (via email or chat) that appears to be a link to an interesting website. You click this link, which results in loading the particular website in your browser. If this website has a page with one or more scripts that access the Gmail APIs to retrieve your data, the lack of something similar to a same-origin policy would allow the script to be executed. Because you're already authenticated to Gmail, and your session data is stored locally on cookies and/or browser local storage, a request to Gmail APIs would submit these cookies as well. So effectively, a malicious website has authenticated to Gmail pretending to be you and is now capable of retrieving any data that the Gmail APIs/services provide.

Now you have a good understanding of the same-origin-policy in web browsers, its importance, and the risks of not having such a policy. But in practice, you still need this to work in certain scenarios, especially with a SPA, to invoke a set of APIs/services, which are outside the domain of the SPA. Let's take a look at how web browsers facilitate resource sharing between different domains to support such legitimate use cases.

4.2.3 Cross-origin resource sharing (CORS)

Web browsers have an exception to the same-origin policy: cross-origin resource sharing (CORS) (<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>), a specification that allows web browsers to access selected resources on different origins. If you are interested in learning more about CORS, we would recommend you have a look at the book CORS in Action: Creating and consuming cross-origin APIs (Manning Publication, 2014) by Monsur Hossain. CORS allows the SPA running on origin localhost:8080 to access resources running on origin localhost:9000. Web browsers use the `OPTIONS` HTTP method along with some special HTTP headers to determine whether to allow or deny a cross-origin request. Let's see how the protocol works.

Whenever the browser detects that it's about to execute a script that makes a request to a different origin, it sends an HTTP `OPTIONS` request to the resource on the particular origin. This request is known as a *preflight request*, which includes the following HTTP headers. You can observe this request by inspecting it on the Network tab of your browser's developer tools.

- `Access-Control-Request-Headers`—Indicates the HTTP headers that the request is about to send to the server (such as `origin`, `x-requested-with`)
- `Access-Control-Request-Method`—Indicates the HTTP method about to be executed by the request (such as `GET`)
- `Origin`—Indicates the origin of the web application (such as `http://localhost:8080`)

The server responds to this preflight request with the following headers:

- Access-Control-Allow-Credentials—Indicates whether the server allows the request originator to send credentials in the form of authorization headers, cookies, or Transport Layer Security (TLS) client certificates. This header is a Boolean value that says true or false.
- Access-Control-Allow-Headers—Indicates the list of headers allowed by the particular resource on the server. If the server allows more than is requested via the Access-Control-Request-Headers header, it returns only what is requested.
- Access-Control-Allow-Methods—Indicates the list of HTTP methods allowed by the particular resource on the server. If the server allows more than is requested via the Access-Control-Request-Method, it returns only the one requested (such as GET).
- Access-Control-Allow-Origin—Indicates the cross-origin allowed by the server. The server may support more than one origin, but what is returned in this particular header is the value of the Origin header requested if the server supports cross-origin requests from the domain of the request originator (such as http://localhost:8080).
- Access-Control-Max-Age—Indicates for how long, in seconds, browsers can cache the response to the particular preflight request (such as 3600).

Upon receiving the response to the preflight request, the web browser validates the response headers to determine whether the target server allows the cross-origin request. If the response headers to the preflight request don't correspond to the request to be sent (perhaps the HTTP method isn't allowed, or one of the required headers is missing in the Access-Control-Allow-Headers list), the browser stops the cross-origin request from being executed.

4.2.4 Inspecting the source that allows cross-origin requests

In the directory chapter04/sample02, open the file located at resource/src/main/java/com/manning/mss/ch04/sample02/rs/ResourceApplication.java with a text editor or IDE. You should find a method named getOrders() in this file and an annotation with the name @CrossOrigin:

```
@GetMapping("/orders")
@CrossOrigin(origins = "http://localhost:8080",
            allowedHeaders = "x-requested-with",
            methods = RequestMethod.GET,
            maxAge = 3600)
public List<Order> getOrders() {
```

Behind the scenes, whenever the server receives an HTTP OPTIONS request to the /orders resource, this piece of code uses the data specified within the @CrossOrigin annotation to build the corresponding HTTP response. By looking at the contents of the annotation, you should be able to figure out how they correspond to the preflight response headers discussed earlier. In the next section, you see how to offload CORS responsibilities to an API gateway layer so that your microservice is relieved from CORS-related duties.

4.2.5 Proxying the resource server with an API gateway

As you saw in section 4.2.4, you need code in your microservice to deal with CORS-specific configurations. According to microservices best practices and recommendations, you should try to avoid burdening the microservice with anything other than what it's primarily designed to do. Another alternative for dealing with the same-origin policy is using an API gateway to act as a reverse proxy to your microservices, which we discuss in this section.

Suppose that you have to expose the same microservice to some other origins someday in the future. In the design discussed earlier, you would need to modify your microservice's `@CrossOrigin` annotation to allow additional origins, which requires rebuilding and redeploying the microservice to expose it to a consumer from a new origin. As mentioned earlier, this modification goes against the microservices recommendations and best practices. Therefore, it would better to use an API gateway solution.

NOTE You can also use any standard reverse-proxy solution such as Nginx (<https://www.nginx.com/>) for the same requirement. We keep mentioning the API gateway because when it comes to separating the consumer [UI] application from the backend, an API gateway becomes useful for much more than CORS negotiations alone.

To examine a practical example of this pattern, look at `sample03` inside the directory `chapter04`. To build the sample, use your command-line client to navigate to the `chapter04/sample03` directory, and execute this command:

```
\> mvn clean install
```

When the build is successful, use the command-line tool to navigate to the `sample03/resource` directory, and execute this command:

```
\> mvn spring-boot:run
```

This command should start the resource server process. To see what kind of resources it provides, execute the following curl command:

```
\> curl localhost:9000/orders
```

You should see a list of orders in JSON format. Next, use the command-line client to navigate to the `sample03/ui` directory, and execute this command:

```
\> mvn spring-boot:run
```

When the process has started successfully, open a private browsing session in your web browser, and go to the following URL:

```
http://localhost:8080
```

You see the message `Login to see your orders`. Click the **Login** button. At this point, open your web browser's developer tools; on the **Network** tab, select the options to preserve/persist logs and disable the cache. On the screen that appears, enter `user` as the

username and password as the password, and submit. You should see an HTML table containing the details of the orders you observed earlier through the curl command.

If you observe the requests that were submitted from your browser, you see no cross-origin requests. All requests are to the same origin as the web application itself (localhost:8080). In the earlier example, the browser made a cross-origin request to the host localhost:9000 to fetch orders, but not in this case. Now the browser makes a request to <http://localhost:8080/orders>. This request is forwarded to your resource server running on localhost:9000 via the embedded API gateway process running within the web application server.

Use a text editor or IDE to open the Java class declared in the file sample03/ui/src/main/java/com/manning/mss/ch04/sample03/ui/UiApplication.java, which is the main class that runs your SPA. Notice that it has an annotation called @EnableZuulProxy. If you went through the samples in chapter 3, you're familiar with this annotation. This annotation instructs the Spring Boot runtime to enable a Zuul API gateway process. Open the file sample03/ui/src/main/resources/application.yml with your text editor or IDE, and notice the following route instruction:

```
zuul:
  routes:
    resource:
      path: /orders/**
      url: http://localhost:9000/orders
```

This route instruction tells the Zuul API gateway to route whatever requests it receives on path /orders/** to URL <http://localhost:9000/orders>. When your browser makes a request to <http://localhost:8080/orders>, the Zuul gateway intercepts this request and forwards it to the URL <http://localhost:9000/orders>.

You have two ways to use an API gateway to handle CORS. One way is to have the API gateway on the same origin as the UI application itself, which is what you tried out in the preceding sample. In this pattern, the UI application (the gateway process running within it) acts as a reverse proxy to your microservice. All requests from the web browser are received by the UI application (the gateway process running within it) and routed to the microservice. Figure 4.4 illustrates this pattern.

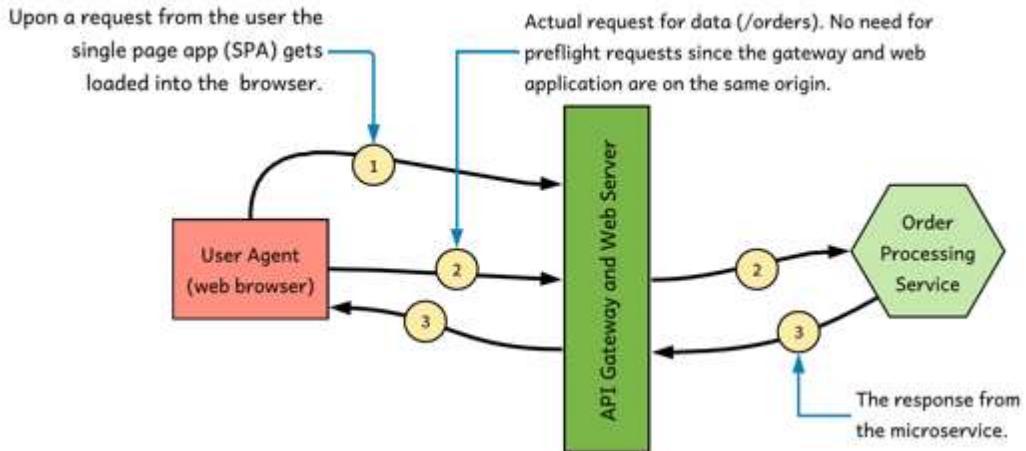


Figure 4.4 API Gateway is running on the same origin as the web application. No preflight request is required. All requests for data are made to the same origin as the web application itself.

In this case, all requests for data are made to the same origin as the web application itself and therefore adhere to the same-origin policy of web browsers. Hence, you don't need to deal with CORS.

The second way to use an API gateway for this problem is to make the API gateway deal with the CORS negotiations and relieve the backend from performing CORS duties. This way, the web browser still performs a cross-origin request, but instead of the microservice, the API gateway handles the preflight request. Figure 4.5 illustrates this pattern.

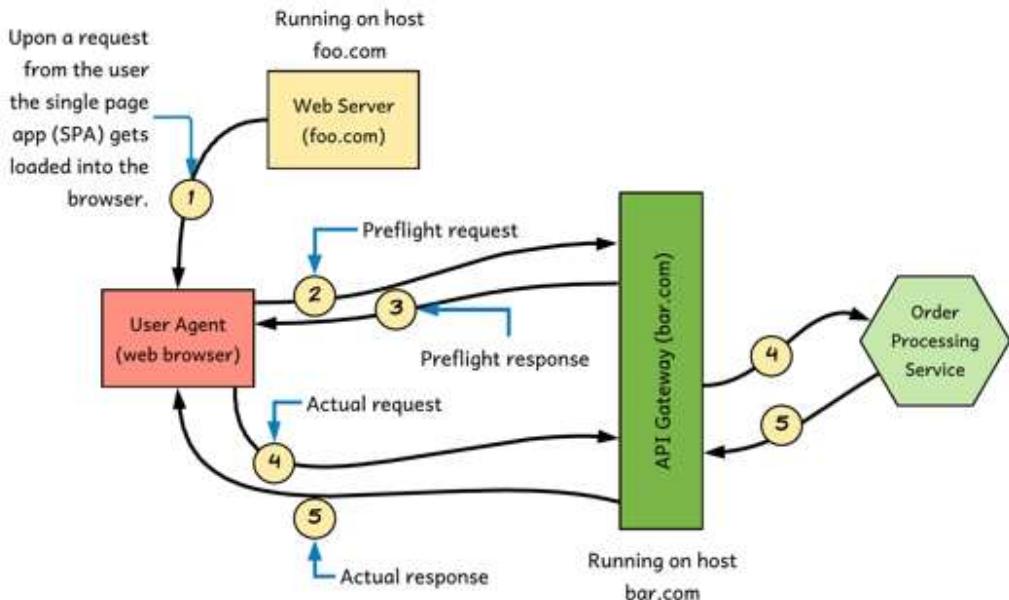


Figure 4.5 API gateway handles preflight requests and running on a different origin from the web application.

4.3 Securing an SPA with OpenID Connect

In this section, you see a practical example of using OpenID Connect to secure your SPA. OpenID Connect is an identity layer built on top of OAuth 2.0. If you are not familiar with OpenID Connect, we recommend you first go through appendix D. In all the preceding examples, you used basic authentication to secure your SPA. In this section, you use OpenID Connect to authenticate users to the SPA. You can find the source code of this example in the chapter04/sample04 directory. Three components participate in this example:

- *The UI server and reverse proxy*—The UI server hosts the SPA and is also an OAuth client with a valid client ID and client secret. You also run a reverse-proxy process on the UI server itself for routing requests to the resource server and authorization server.
- *The OAuth 2.0 authorization server*—This component that plays the role of the authorization server, which also is the OpenID Connect provider hosting the /userinfo endpoint. An OpenID Connect client to retrieve further information about the authenticated user can use the userinfo endpoint. In practice, this endpoint is rarely used.
- *The resource server*—This server hosts the details of your orders and exposes data via APIs. The UI application fetches the details to display from this resource server.

To build the source code of these samples, navigate to the chapter04/sample04 directory in your command-line client, and execute this command:

```
\> mvn clean install
```

When the build is successful, use the command-line client to navigate to the directories `sample04/authserver`, `sample04/resource`, and `sample04/ui` one by one (each using a different terminal window), and execute this command under each directory:

```
\> mvn spring-boot:run
```

After each execution of the command (under all three directories), wait for the processes to complete successfully. Then open a private browsing session in your web browser, and go to the following URL.

```
http://localhost:8080/
```

You should see the message `Login to see your orders`. Open the developer tools of your web browser; on the **Network** tab, select the options to preserve/persist logs and disable the cache. Then click the **Login** button. You're redirected to the login page of the authorization server (`http://localhost:9999/uaa/login`). Notice the requests on the Network tab of the browser's developer tools.

The first request is a request to the login page on the UI server itself (`http://localhost:8080/login`). This request results in a 302 redirect response. If you notice the `Location` header in the response message, you should see it is sending a redirect to the `/authorize` endpoint of the authorization server with the client ID as a request parameter:

```
Location:  
http://localhost:9999/uaa/oauth/authorize?  
client_id=acme&  
redirect_uri=http://localhost:8080/login&  
response_type=code&  
state=F3jdkW
```

Upon receiving this redirect response, the browser makes a GET request to the URL received in the `Location` header. That request also results in a 302 redirect, this time with a redirect to the login page of the authorization server:

```
Location: http://localhost:9999/uaa/login
```

The browser makes a GET request to this URL, and that request results in what you see on the login page of your browser. When you clicked the Login button on the web page, your browser made a request to the `/login` resource of your UI server. The UI server, as an OAuth2.0 client application, initiated an OAuth2.0 authorization code grant flow that resulted in the authorization server's presenting the login page. The `client_id` and the `client_secret` of the OAuth2.0 client application (UI server) are embedded in its source, which runs on the server. You can find this detail in the file `sample04/ui/src/main/resources/application.yml`. If you search for the strings `clientId` and `clientSecret`, you should be able to find this information. Figure 4.6 illustrates the sequence of events.

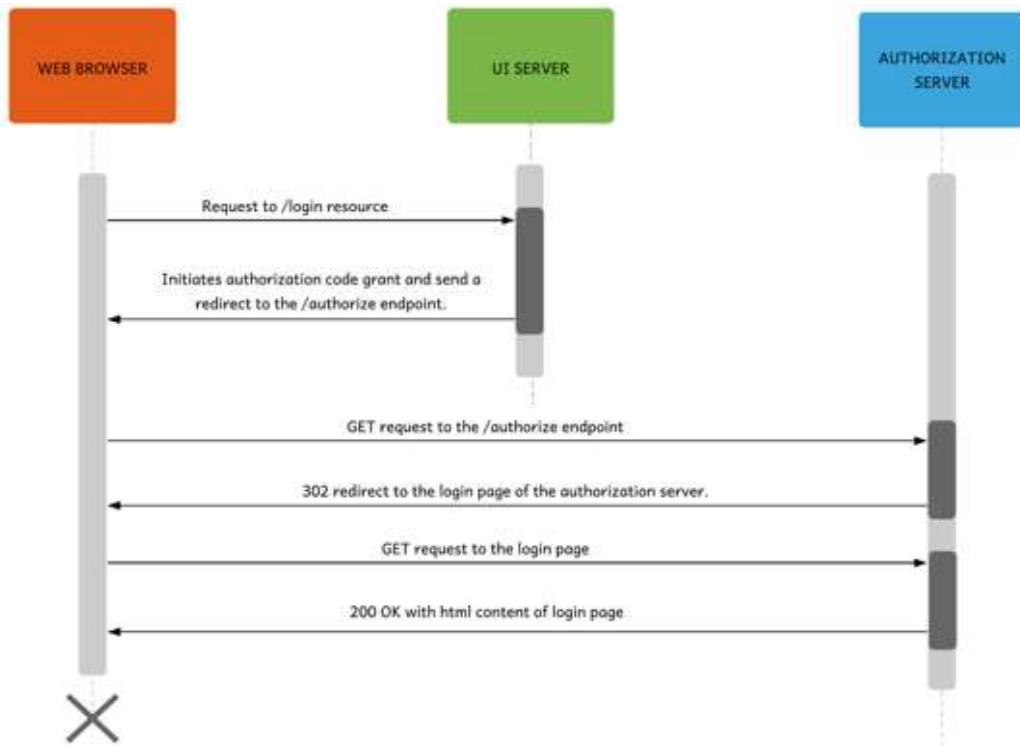


Figure 4.6 Sequence of events that present the login page of the authorization server during an OpenID Connect login flow.

Next, you provide the username and password to authenticate yourself. Few more request-and-response exchanges will happen beyond this point, and the **Network** tab of your browser's developer tools can become a bit noisy, so before proceeding, it's OK to clear the **Network** tab (not to close the developer tools) so that you can easily focus on the next set of request responses.

Enter `user` as the username and `password` as the password, and submit. You should see a consent page where the authorization server requests your consent to issue an authorization code with scope `openid` to the application `acme` (`client_id`) running at `http://localhost:8080/login`. If you observe the Network tab of your browser's developer tools after submitting, you see a POST request to the `/login` endpoint of the authorization server (`http://localhost:9999/uaa/login`). This request submits the username and password that you entered. The authorization server responds with a 302 redirect, with the location set to the `/authorize` endpoint of the authorization server:

`Location:`
`http://localhost:9999/uaa/oauth/authorize?`

```
client_id=acme&
redirect_uri=http://localhost:8080/login&
response_type=code&
state= F3jdkW
```

Below the response headers, you should notice a header named `Set-Cookie` being sent from the authorization server:

```
Set-Cookie:
JSESSIONID=5393AAD882E08A6A4A4DE1B50237E0C9;
Path=/uaa;
HttpOnly
```

This response sets an `HttpOnly` (<https://www.owasp.org/index.php/HttpOnly>) cookie on your web browser against the `/uaa` path of your authorization server. This cookie is set to indicate that you have a valid authenticated session in your browser with the authorization server.

After receiving this response, notice on the Network tab that the browser makes a `GET` request to the `/authorize` endpoint of the authorization server (as instructed by the `Location` header). If you inspect the request headers of this request, you see that the browser submits the preceding cookie as well:

```
Cookie: JSESSIONID=5393AAD882E08A6A4A4DE1B50237E0C9;
```

When this request is received by the `/authorize` endpoint of the authorization server, it knows that the user is already authenticated into the server (through the cookie); therefore, instead of displaying the login page, it presents the consent page for issuing an authorization code (see figure 4.7).

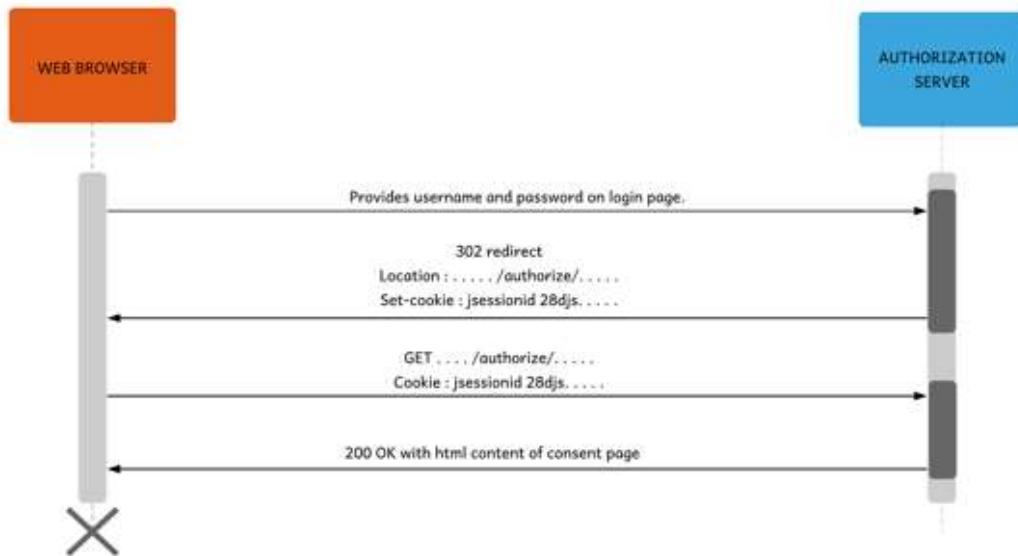


Figure 4.7 Sequence of events that happen while log in to the authorization server during an OpenID Connect login flow.

When you complete these tasks, you can clear the Network tab and provide your consent by clicking the Approve button. Be aware of session timeouts, though. If you spent a long time studying the request-and-response flows, your session may have timed out by the time you click the Approve button, and you may get an error page. In that case, close the current browser window, open a new private browsing session, and retry the steps to this point. If you're successful, you should see the list of orders on your web page.

Next, your browser makes an HTTP POST request to the /authorize endpoint of the authorization server. If you observe the request payload on the **Network** tab of your browser's developer tools, you see a parameter indicating that you approved the consent as shown below.

```
user_oauth_approval: true
```

The browser also submits the cookie so that the authorization server can track your session. Upon receiving this request, the authorization server responds to the browser with a 302 redirect that points to the login page of the UI server. It also provides the authorization code as a query parameter as shown below.

```
Location: http://localhost:8080/login?code=FjzX6f&state=F3jdkW
```

The browser next submits a GET request to the UI server on the URL received on the preceding Location header. Upon receiving this request, the UI server extracts the authorization code

and uses it to make a `/token` request to the authorization server for an access token. This `/token` request happens on the server side (UI server) and isn't visible in the web browser's developer tools. You can observe this request if you use a network monitoring tool such as Wireshark (<https://www.wireshark.org/>). Following is what the `/token` request looks like (captured by a network monitoring tool), with the payload displayed in decoded format for readability.

```
POST /uaa/oauth/token HTTP/1.1
Authorization: Basic YWNtZTphY21lc2VjcmV0
Accept: application/json, application/x-www-form-urlencoded
Content-Type: application/x-www-form-urlencoded
Host: 127.0.0.1:9999

grant_type=authorization_code&
code=FjzX6f&
redirect_uri=http://localhost:8080/login
```

The UI server receives the following response, which contains the access token information. The access token and refresh token are JWTs (self-contained tokens) that have been modified and shown below for readability purpose:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked

{"access_token":"u8jah29S.ahU82jN.hu8kiw",
 "token_type":"bearer",
 "refresh_token":"hu8YiK.N2Qry.P0ti4N",
 "expires_in":43199,
 "scope":"openid",
 "jti":"6f12f5af-a15c-40b6-9020-2c3b87e007ad"}
```

After receiving the token successfully, the UI server responds to the browser with a 302 redirect pointing to the home page of the web application as shown below.

```
Location: http://localhost:8080/
```

If you notice the response headers you will also notice that there is a cookie set on the web browser using a Set-Cookie header.

```
Set-Cookie: JSESSIONID=D14240CDB27CA553A4E08510A0B99D56; Path=/; HttpOnly
```

This aspect is an important one to understand. As you may have noticed, the UI server obtained an access token from the authorization server, but this access token isn't given to the web browser (SPA) at any point. What is given is a cookie. The session between the browser and the UI server are tracked by means of a cookie, not the OAuth 2.0 token. Figure 4.8 illustrates the sequence of events.

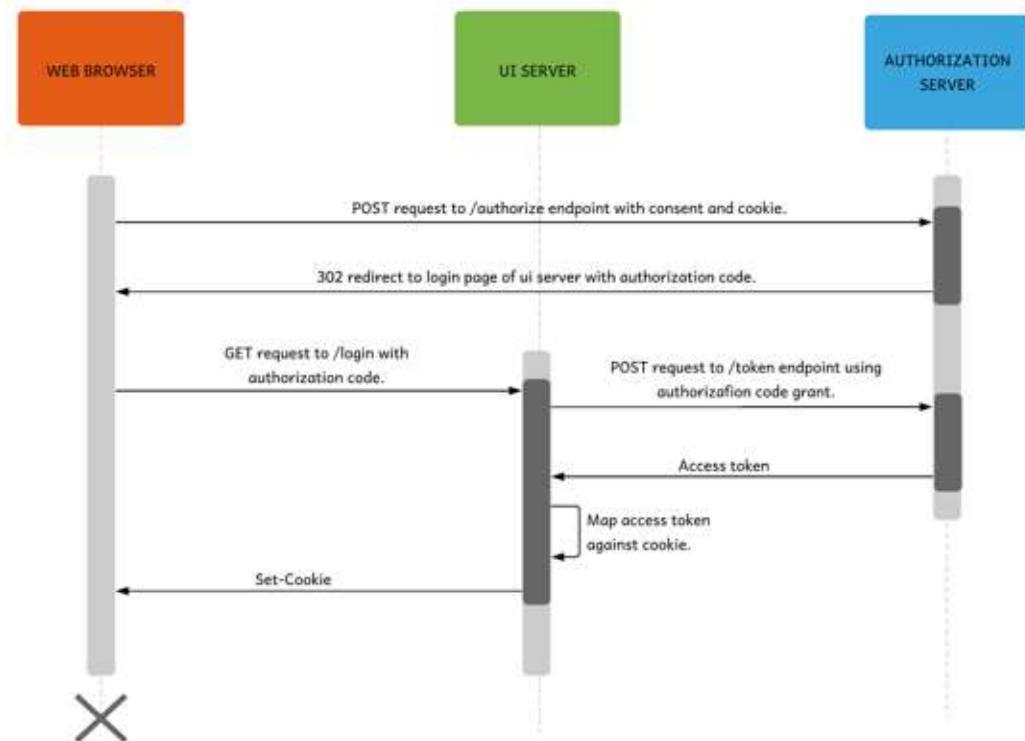


Figure 4.8 Sequence of events that happen while obtaining an access token from the authorization server during an OpenID Connect login flow.

You may wonder why you have cookies coming into action instead of the direct access token. The reason, as we discussed in chapter 3, is that JavaScript-based SPAs aren't capable of protecting sensitive user information in the application itself (web browser). Whatever information is stored in the browser in a form that's accessible by JavaScript is vulnerable to theft in case your application becomes vulnerable to XSS attacks. To minimize the risk of such attacks, use an `httpOnly` cookie to represent the access token. This way, this cookie becomes inaccessible by JavaScript code, but it is submitted in every request being made to the UI server. The UI server can pick the access token that corresponds to the cookie and make requests to the resource server by using the relevant access token.

4.3.1 Where does OpenID Connect fit in?

So far we have discussed how your SPA along with the UI server makes an OAuth2.0 handshake with the authorization server and how to obtain an access token. If you continue observing the **Network** tab of your browser's developer tools, you see that after a successful login, the browser makes several **GET** requests to various documents (JavaScript, CSS, and so

on), all of which result in 200 OK responses. Before the last request from the browser, you should see a request to the `http://localhost:8080/user` resource of the UI server, along with the cookie that was set. The SPA at this point tries to fetch information about the logged-in user to display the details of the orders, which is where OpenID Connect comes into play.

You can observe the above behavior in the source code if you open the file at `sample04/ui/src/app/app.component.html`, which has the following code block. This code displays the order information only to an authenticated user:

```
<div [hidden]="!authenticated">
  <p>{{ orders.length }} Orders in Queue</p>

  <table cellspacing="5px" style="border:1px solid black;" border="1">
    . . .
  </table>
</div>
<div [hidden]="authenticated">
  <p>Login to see your orders</p>
</div>
The value of the variable authenticated is set by executing a function named authenticate(),
located in the file sample04/ui/src/app/app.component.ts. This function calls the /user
resource on the UI server. The reverse proxy running on the UI server routes this
request to the /userinfo endpoint of the authorization server. This route is specified
in the file sample04/ui/src/main/resources/application.yml:
zuul:
  routes:
    user:
      path: /user/**
      url: http://localhost:9999/uaa/user
```

The UI server, also being an OAuth 2.0 client, looks up the corresponding access token bound to the cookie it received in the request and constructs the correct request to be sent to the userinfo endpoint of the authorization server. The request looks like this:

```
GET /uaa/user HTTP/1.1

accept: application/json, text/plain, /*
referer: http://localhost:8080/
authorization: bearer u8jah29S.ahU82jN.hu8kiw
x-forwarded-host: localhost:8080
x-forwarded-proto: http
x-forwarded-prefix: /user
x-forwarded-port: 8080
x-forwarded-for: 0:0:0:0:0:0:0:1
Host: 127.0.0.1:9999
```

You can get this code from a network-monitoring tool. Because the UI server initiates this request, you can't capture it in the browser's developer tools. This request includes the access token you obtained before, provided in the form of an authorization bearer header. Upon receiving this request, the authorization server sends the user information to the UI server, which passes it along to the browser as is. The browser (SPA) uses this information to determine the whether the logged-in user is authenticated and then renders the required

information based on that status. Figure 4.9 shows how the UI determines the state of the logged-in user.

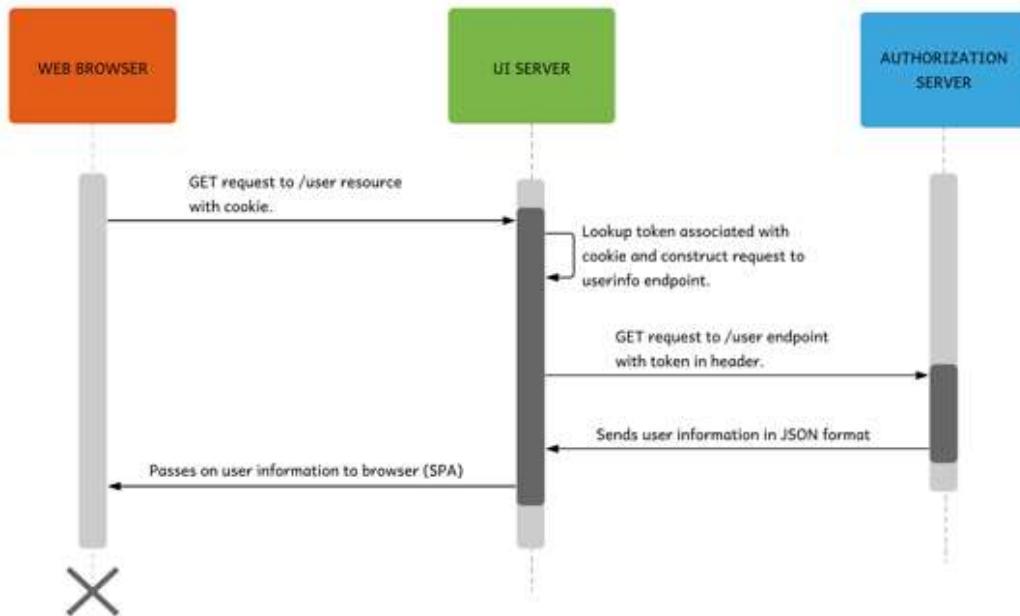


Figure 4.9 Obtaining user information through OpenID Connect.

This way, the SPA can obtain the user information of the logged-in user. This information can be used for whatever purpose is required, such as rendering certain actions (links) based on user privileges, where the privileges are obtained as part of the user info response message.

4.4 Federated authentication

In the preceding sections, we discussed building a SPA and securing it with OpenID Connect. Something that we didn't explicitly talk about but that happened behind the scenes was that the resource server was also secured with OAuth 2.0. We discussed securing your resources with OAuth 2.0 in chapters 2 and 3, and we used the same mechanism to protect your resource server in this chapter. Figure 4.10 shows at a high level how the resource server is protected.

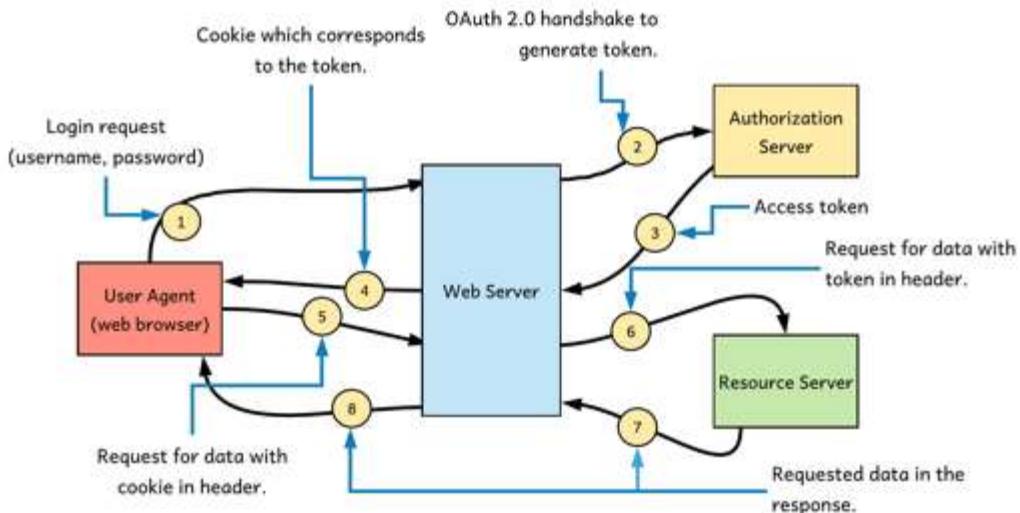


Figure 4.10 End-to-end authentication flow between SPA and resource server. Users authenticate to SPA with OpenID Connect and the resource server protects all its APIs with OAuth 2.0.

In sample04, you had the authorization server issue self-contained access tokens in the form of signed JWTs (which is a JWS). As we discussed in chapter 3, when the resource server (or the API gateway that sits in front of it) receives a self-contained access token, it doesn't have to talk to the authorization server to validate it. The resource server first validates the authenticity (proof of identity) (<https://www.brighthub.com/computing/smb-security/articles/31234.aspx>) of the access token by verifying its signature and then checks the validity of its contents (intended audience, expiry, scopes, and so on). To validate the authenticity of the access token (JWT), the resource server requires the public key of the authorization server that signed the token. You can see this value under a key named `security.oauth2.resource.jwt.keyValue` in the `chapter04/sample04/resource/src/main/resources/application.properties` file.

As you may have noticed, a trust relationship is required between the resource server and the authorization server. Even if you used an opaque token, so that the resource server would need to talk to the authorization server that issued the token to validate it, you still need a trust relationship between the resource server and the authorization server.

NOTE Although we keep using the term **resource server** in this chapter, you could use an API gateway. We're omitting a discussion of the API gateway to reduce the number of components discussed in this chapter.

The client application (SPA), authorization server, and resource server need to be in the same trust domain. Suppose that in your online retail store, you have to work with a third-party shipping service to make shipping requests for the orders that are placed and to get details on

pending shipments. When a user logs in to your retail store, he authenticates against your trust domain. But when he inquires about the status of his shipments, your web application needs to talk to a third-party shipping service. The shipping service needs to make sure that it gives details only to the relevant users (user A can't get details on orders placed by user B) and therefore needs to know the authenticity of the user who's logged in to the system. Because the shipping service is in a separate trust domain, by default it won't know how to validate a token issued by your own private trust domain.

4.4.1 Multiple trust domains

If a client application works across multiple trust domains, the preceding solution won't work. When a resource server is presented a token issued by an authorization server of another trust domain, it won't be able to validate the token.

If the presented token is a self-contained token in the form of a signed JWT, the resource server needs to know the public certificate used by the authorization server in the foreign trust domain to validate the signature of the token. Because the authorization server and resource server are separated by trust domain, it's unlikely that the resource server will know this information.

If the resource server was presented an opaque token, it would need to contact the authorization server that issued the access token to validate it. Again, because these two servers are separated by trust domain, it's unlikely for this scenario to be possible. The resource server may not know where the relevant authorization server resides, and even if it does, it may not have the necessary credentials to talk to its token validation endpoint. Figure 4.11 illustrates the trust boundaries between systems.

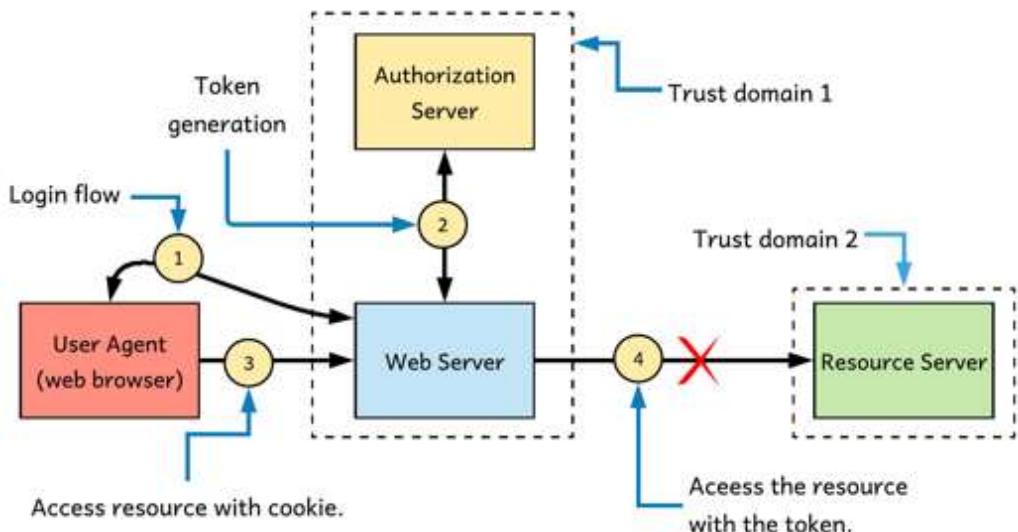


Figure 4.11 The authorization server and resource server are in two trust domains. The resource server in trust domain 2 does not know how to validate a token issued by the authorization server in trust domain 1.

As shown in figure 4.11, step 4 will fail because the token that's passed to the resource server at this point is obtained from trust domain 1. Because the resource server is in trust domain 2, it won't know how to validate the token that it receives; hence, the request will fail with an authentication error.

4.4.2 Building trust between domains

To solve this problem, you need to build a mechanism that builds a trust relationship across domains. An authorization server itself usually defines a trust domain. A given trust domain has many resource servers, web applications, user stores, and so on, but it has only one authorization server that governs how each component or application in the particular domain should authenticate. It is the single source of truth. Therefore, you need to build a chain of trust between the authorization servers of each domain. This chain of trust combined with the token exchange pattern (using the JWT grant type) can provide a solution to your problem (see figure 4.12).

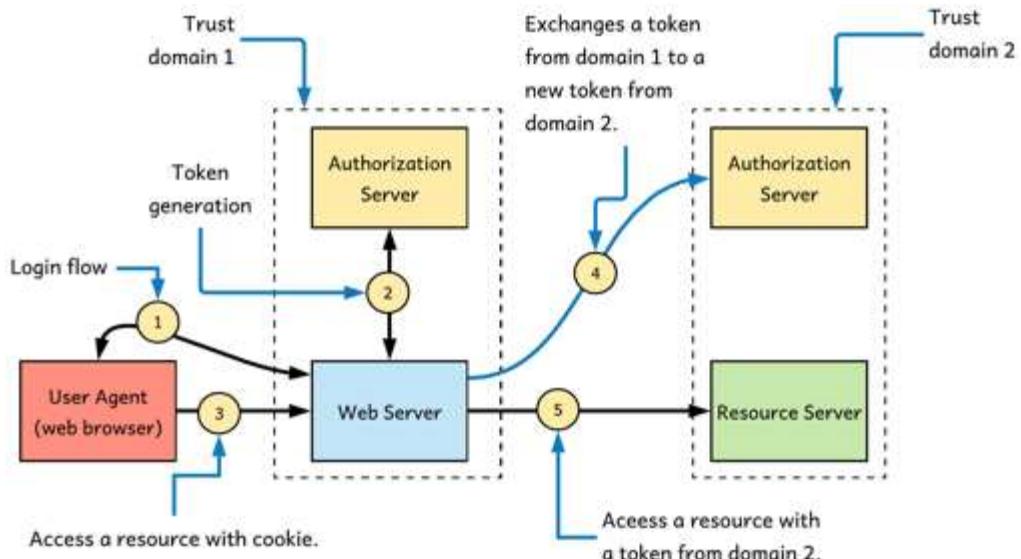


Figure 4.12 Token exchange pattern to obtain a token valid for domain 2. The authorization server in domain 2 trusts the authorization server in domain 1, but the resource server in domain 2 only trusts the authorization server in its own domain.

As shown in figure 4.12, before the service provider (web server) attempts to access resources from the resource server in domain 2, it should perform a token exchange from the authorization server in domain 2. Assuming that the token obtained from the authorization server in domain 1 is a JWT (not an opaque token), it can use the JWT bearer grant type (<https://tools.ietf.org/html/rfc7523>) to request an access token from the authorization server in domain 2.

The JWT bearer grant type (officially known as the JWT profile for OAuth2.0 client authentication and authorization grants) has a simple protocol. It accepts a JWT and performs the relevant validations on the token (verifying its signature, expiry, and so on); if the token is valid, it issues a valid OAuth2.0 token to the client, which can be another JWT itself or even an opaque token string. The important thing in this scenario is the ability of the authorization server in domain 2 to validate the token (JWT) it receives and to exchange it for a new access token. For this purpose, you need to build a trust relationship between the authorization servers in domain 1 and domain 2. Building this trust relationship could be a matter of exchanging public certificates.

4.5 Summary

- Single page applications perform better since it reduces the network chattiness by performing all rendering on the web browser and since it reduces the workload on the

web server.

- The SPA architecture brings simplicity to microservice architectures since they do not require complex web application hosting facilities such as JBoss, Tomcat and so on.
- The SPA architecture abstracts out the data layer from the user experience layer.
- There are security restrictions and complexities in SPAs due to the same origin policy on web browsers.
- The same-origin policy is a concept in web browsers introduced to ensure that scripts running on a particular web page can make requests only to services running on the same origin.
- The same-origin policy applies only to data access, not to CSS, images, and scripts, so you could write web pages that consist of links to CSS, images, and scripts of other origins.
- OpenID Connect is an identity layer, which is built on top of OAuth 2.0. Most SPA use OpenID Connect to authenticate users.
- Since SPAs may consume APIs (data-sources) from multiple trust domains a token obtained from one trust domain would not be valid for another trust domain. We need to build token exchange functionality when an SPA hops between multiple trust boundaries.

5

Engaging throttling, monitoring, and access control

This chapter covers

- How throttling works and why we need to apply throttling to our microservices
- Different dimensions of throttling and their impact
- Setting up Zuul as an API gateway for throttling
- An introduction to observability and its importance in a microservices deployment
- Using Prometheus and Grafana to monitor our microservices
- Applying access control policies via Zuul and Open Policy Agent (OPA)

In chapter 3 we introduced the API Gateway architectural pattern and discussed its applicability in a microservices deployment. Zuul is an open source API gateway developed by Netflix to front all its microservices. Zuul provides dynamic routing, monitoring, resiliency, security and more. It acts as the front door to Netflix's server infrastructure, handling traffic from Netflix users around the globe. In chapter 3 we discussed how to enforce security based on OAuth 2.0 for your microservices using Zuul as the API gateway. In this chapter we extend those samples to use Zuul to handle throttling, apply access control policies and we also discuss the monitoring aspects in a microservices deployment.

5.1 Throttling at the API gateway with Zuul

In this section, we discuss the types of threats a typical microservices deployment is exposed to by allowing too many requests within a particular time frame, and why it is important to throttle requests. Take a look at figure 5.1 to refresh our memory from chapter 3 on the participants of an API Gateway architecture pattern.

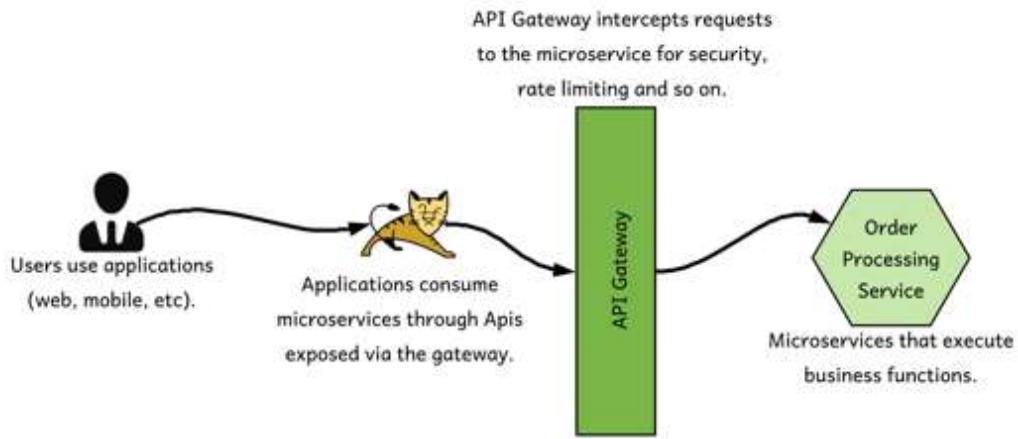


Figure 5.1 API gateways are used in microservice architectures to expose microservices for external consumption. Users use applications for various tasks and these applications consume services exposed by API gateways.

As you can see from figure 5.1, the client application accesses the API gateway, which in turn sends the requests to your target microservices (back-end). As per the diagram, the API gateway and the target service are both request serving nodes in the system. Every request sent by a client application needs to be processed by the API gateway and every valid request received by the API gateway needs to be sent to the target microservice for processing. As requests from the client applications increase, it directly impacts the performance and scalability factors of the API gateway. The increased number of valid requests, which are authenticated and authorized, have a direct impact on the performance and scalability factors of the target microservices. The way we usually deal with a rising number of requests to process is to scale up the application layers (API gateway and target services). Scaling up has two primary models: vertical scaling and horizontal scaling. Vertical scaling is a mechanism of increasing the computing power on which our software runs, such as increasing the memory and CPU capacity of the corresponding servers. With horizontal scaling, we scale our deployment by adding more VMs (or computers) to our pool of resources where our software (microservices) run and execute all of them in parallel.

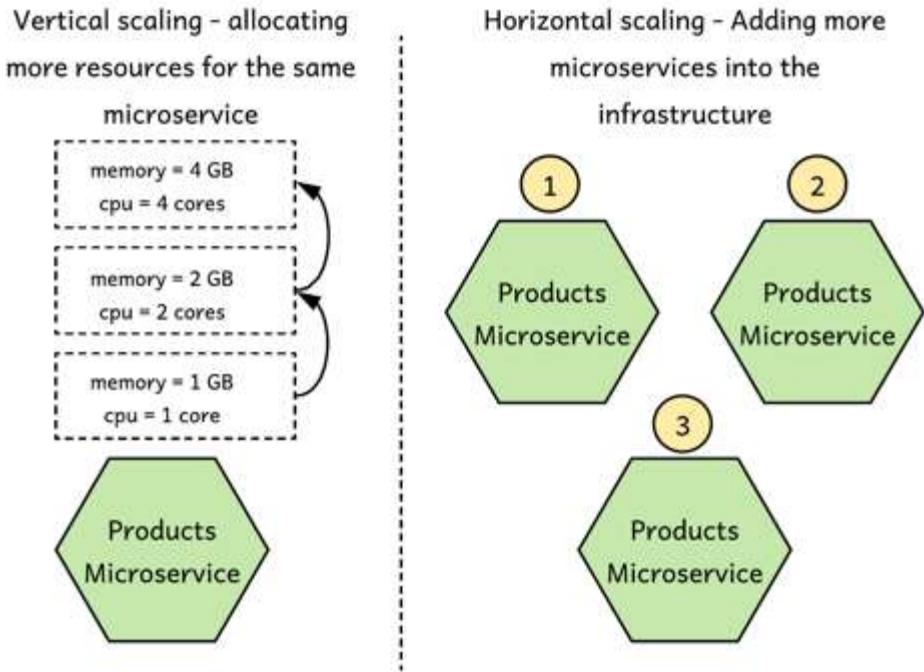


Figure 5.2 Vertical scaling vs. horizontal scaling. The resources available on the host of the microservice limit vertical scaling. Horizontal scaling in theory is unlimited.

Scaling doesn't come for free. Whatever mode of scaling we opt for, it comes at a cost. There is also a physical limit on how much we can scale our systems. This limit is sometimes defined by the amount of available physical hardware. And at times this is also limited by the ability of our software to scale horizontally. Especially when it comes to stateful software systems, such as a system that depends on a relational database, scaling the software beyond a certain level becomes harder and harder due to the complexities involved in scaling the relational database. All of these complexities lead us to a point where it becomes necessary for us to limit the number of requests being served by our system. And we need to do it on a fair premise so that all consumers of our services get a fair quota of the total amount of traffic we can handle. Limiting the number of requests can also help to protect our systems from attackers who attempt to perform denial of service type of attacks by sending in a huge number of requests and thus preventing the others from gaining access to the system as a consequence. Let us now look at the various types of throttling we can apply on our systems.

5.1.1 Quota based throttling for applications

In this section we discuss about how we can apply throttling to our microservices based on request quotas allocated to specific client applications. Allocating request quotas become useful

in cases such as monetizing microservices and also in cases where we want to prevent microservices being consumed more than its capacity allows.

As in figure 5.3, some sort of application or device consumes the Order Processing microservice. It is also possible that more than one application gain access to the same microservice, such as the Order Processing microservice being accessed by a mobile device, a web application, and a desktop application. When the number of consumer applications of the Order Processing microservice increases, the higher the impact it has on the Order Processing microservice. Assuming for argument's sake that the load the Order Processing service can handle is 1000 requests per second, if all the mobile applications collectively consume the full 1000 requests, it will starve the web application and the desktop application (figure 5.3).

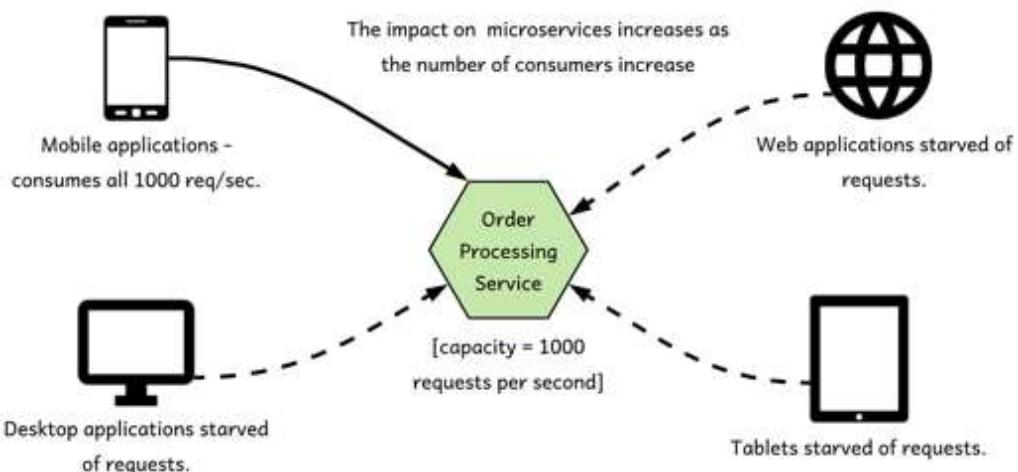


Figure 5.3 The more consumers a microservice has, the greater the capacity it must handle

The way to solve this problem is to provide each application a quota, such as 200 requests per second, and to enforce this quota at the API gateway. This way a given application or device would not be able to starve the rest of the consumers in the system. The maximum capacity it can consume from the system would now be limited to 200 requests per second. This leaves room for other consumers to consume the remaining 800 requests within a given second. Hence eliminates the risk of a single application causing a denial of service attack.

For this mechanism to work, for each request the API gateway processes, it needs to be able to identify the application from which it was originated. In case it is unable to identify the application or device type from which the request originated, it should default to a common value, such as say 100 requests per second. Assuming the APIs on the API gateway are secured using OAuth2.0, the consumer applications would need to have a unique client ID each as we discussed in chapter 3. When an application sends an access token to the API gateway the gateway can then introspect the access token (by talking to the OAuth authorization

server) and retrieve the corresponding client ID for which the token was issued to. This client ID can be used as the mechanism of uniquely identifying the application from which the request was originated. The API gateway would then count the number of requests being served within a time window of 1 second against each unique client ID. When a given client ID goes beyond 200 requests per second, the API gateway would prevent further requests from that client ID from being sent to the target services until the time window has passed.

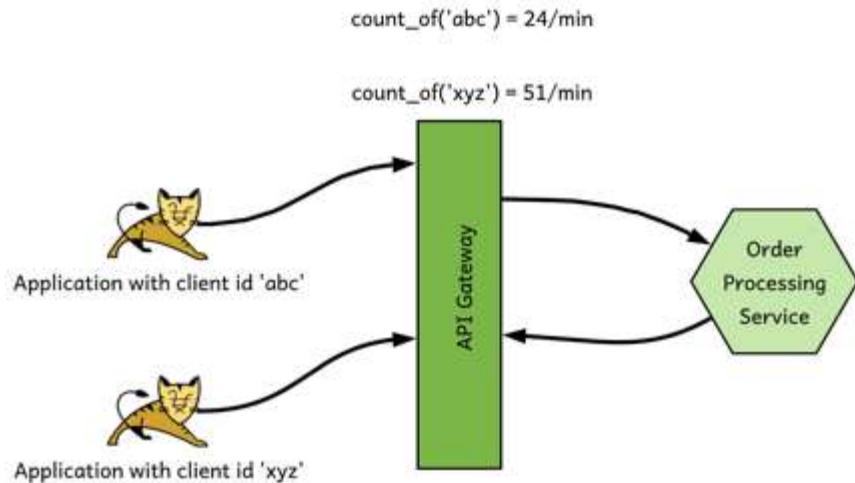


Figure 5.4 The gateway keeps a count of the requests made by each application within the respective time windows. The ID of the application (client_id in the case of OAuth2) is used as the unique identifier of the application.

5.1.2 Fair usage policy for users

In this section we discuss how we can ensure all the users of applications are treated equally. Or in other words certain users of an application do not get denied access to a service due to some other users consuming larger chunks of the quota.

As we discussed in section 5.1.1, we would need to apply a limit of requests for a given time window for each application. This is to prevent one or few applications consuming a majority of the capacity of our services, thus resulting in a denial of service for other applications. The same problem could occur for users of applications. Say for example, an application has 20 users and the application is given a quota of 200 requests per second. If a given user consumes all these 200 requests within a time window of 1 second, the other users will not be left with anything to consume. Thus, resulting in a denial of service for those users. It is therefore important to impose a fair usage policy on an application to ensure all users get a fair share of the quota given to the application.

Inspecting the user's credential used to access the API gateway/microservice helps identifying the user to apply fair usage policies. For example, if we use Basic Authentication to

protect our APIs, then we can have username as the user identifier. If we use self-contained access tokens or JSON Web Tokens (JWT) to protect the APIs, then we can have the `sub` claim that comes with the JWT as the user identifier. If the client application uses regular OAuth 2.0 tokens (or the reference tokens), then we can find the corresponding user identifier by introspecting the token by talking to the authorization server.

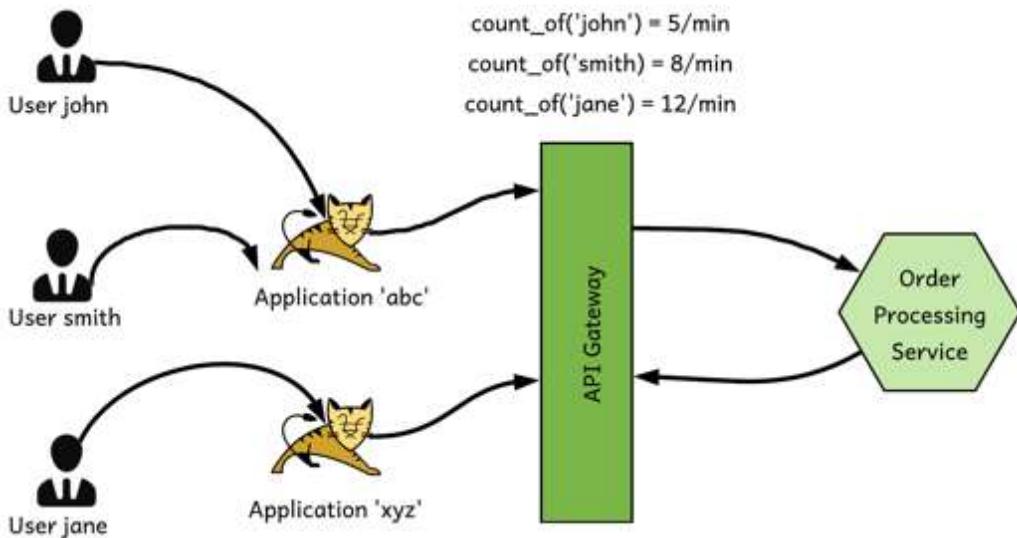


Figure 5.5 The gateway ensures a fair usage is maintained across all users of the application so that one (or few) users of the application cannot starve the other users of the application.

5.1.3 Running the sample to apply quota based throttling

Before we discuss further details on throttling let's run through an exercise which gives us an idea of what it means to throttle requests by users. To begin, download the chapter 5 samples from GitHub (<https://github.com/microservices-security-in-action/samples>) to your computer.

NOTE The examples in this chapter use Java version 11, but still should work with Java 8+. Before executing the examples in this chapter, make sure that you've stopped running the examples from other chapters or elsewhere. You could experience port conflicts if you attempt to start multiple microservices on the same port.

Once you've downloaded all the samples from GitHub repository, you should see a directory called `sample01` inside the `chapter05` directory. You will see three sub-directories within the `sample01` directory.

- The `oauth2_server` directory contains the code of the authorization server. This is the same authorization server we used in chapter 3.
- The `gateway` directory contains the source of the Zuul gateway, which applies throttling

policies on our services. Just as how it intercepted requests and applied security (in chapter 3), this gateway now performs throttling as well.

- The `service` directory contains the source code of the microservice, which actually hosts the business logic.

Navigate to the `oauth2_server` directory from your command line client tool and execute the following command to build it.

```
\> mvn clean install
```

Once the code is successfully built, execute the following command to run it.

```
\> mvn spring-boot:run
```

Execute the same above two commands from the `gateway` and the `service` directories, but from different terminal tabs each, to build and run the Zuul gateway and the Order Processing microservice. Once all three processes are up and running we can start using them to see what it looks like to rate limit our services.

First let's get an access token to access the Order Processing microservice. We expose the Order Processing microservice via the Zuul API gateway, and it enforces OAuth 2.0 based security. So, we need a valid OAuth 2.0 access token to make requests to the API gateway. You can get an access token from the authorization server (via the gateway) by executing the following command from you command line client.

```
\> curl -u application1:application1secret -H "Content-Type: application/x-www-form-urlencoded" -d "grant_type=client_credentials" 'http://localhost:9090/token/oauth/token'
```

If the request is successful you should get a response similar to below.

```
{"access_token": "c0f9c2c1-7f81-43e1-acac-05af147662bb", "token_type": "bearer", "expires_in": 3599, "scope": "read write"}
```

Once you have the token we can access Order Processing microservice via the Zuul API gateway. Let's run the following cURL command to do an HTTP POST to the Order Processing microservice. Make sure to use the value of the same access token you received from the previous request. If not, you will receive an authentication failure error.

```
\> curl -H "Authorization: Bearer c0f9c2c1-7f81-43e1-acac-05af147662bb" -d "[{"items": [{"itemCode": "IT0001", "quantity": 3}, {"itemCode": "IT0004", "quantity": 1}], "shippingAddress": "No 4, Castro Street, Mountain View, CA, USA"}]" -H "Content-Type: application/json" http://localhost:9090/retail/orders -v
```

If the request is successful you get a response as below with the status code 200. This indicates that the Order Processing microservice has successfully created the order. The `orderId` in the response is the unique identifier of the newly created order. Take note of the `orderId` since we need it in the next step.

```
{"orderId": "cabfd67f-ac1a-4182-bd7f-83c06bd4b7bf", "items": [{"itemCode": "IT0001", "quantity": 3}, {"itemCode": "IT0004", "quantity": 1}], "shippingAddress": "No 4, Main Street, Colombo 1, Sri Lanka"}
```

Next let us use the `orderId` to query our order information as below. Make sure to use the same access token and the same `orderId` as before. In our request the `orderid` is the `cabfd67f-ac1a-4182-bd7f-83c06bd4b7bf`.

```
/> curl -H "Authorization: Bearer c0f9c2c1-7f81-43e1-acac-05af147662bb"
    http://localhost:9090/retail/orders/cabfd67f-ac1a-4182-bd7f-83c06bd4b7bf -v
```

You should see a response with the status code 200 including the details of the order we placed before. Execute the same request for 3 more times. You should observe the same response, with status code 200 being returned. If you execute the same request for the 4th time within a minute you should now see a response with the status code 429 saying the request is throttled out. The duration of the time window (1 minute) is configured in a Java class that we will take a look at shortly. The response would look like below.

```
< HTTP/1.1 429
< X-Application-Context: application:9090
< Transfer-Encoding: chunked
< Date: Mon, 18 Mar 2019 13:28:36 GMT
<
>{"error": true, "reason":"Request Throttled."}
```

In chapter 3 we used the API gateway to enforce security on our microservices. We extend the same gateway here to enforce throttling limits as well. In chapter 3 we used a Zuul filter on our gateway to enforce security. Here we introduce another filter that executes after the security filter to enforce throttling limits.

Let's take a look at the code. The Java class file that implements throttling is `ThrottlingFilter.java` (listing 5.1), which you can find inside the `gateway/src/main/java/com/manning/mss/ch05/sample01/gateway/filters/` directory. If you open this file in a text editor or in our IDE, you will see 3 methods on this class named as `filterType()`, `filterOrder()` and `shouldFilter()`.

Listing 5.1. A code snippet extracted from `ThrottlingFilter.java`

```
public String filterType() {
    return "pre";      #A
}

public int filterOrder() {
    return 2;        #B
}

public boolean shouldFilter() {
    return true;     #C
}
```

#A The filter-type pre indicates that this filter should execute before the request being processed.

#B The filter-order 2 indicates that this filter executes after the OAuthFilter, who's filter-order is 1

#C The should-filter true indicates that this filter is active.

At the top of the `ThrottlingFilter.java` class we initialize a counter of the `CounterCache` type, which is an in-memory map. Each entry in this map holds a counter against its key. And each entry in the map resides for approximately 60 seconds, and after that it is being removed.

```
//Create a counter cache where each entry expires in 1 minute and the cache is cleared every
10 seconds.
//Maximum number of entries in the cache would be 10000.
private CounterCache counter = new CounterCache(60, 10, 10000);
```

The key in this map is quite important and it is what we count our requests against. In this particular example we use the access token as the key to count against. Since the access token itself is kind of a secret (or like a password), possibly you can use the hashed value of it rather using it as it is. If you make 2 requests using the same access token within a 60 second time window, the counter of that token would be 2.

```
//Get the value of the token by splitting the Authorization header
String key = authHeader.split("Bearer ")[1];
Object count = counter.get(key);
```

We can similarly use any other key or even multiple keys to count against, depending on our use cases. For example, if we want to count the number of requests of an application, we would use the `client_id` as the key to count against. We can also use any attribute such as the user's username, IP address, OpenID Connect claims and so on as a key.

5.1.4 Maximum handling capacity of a microservice

The gradual increase in applications and users demands the capacity or the maximum number requests a microservice can handle. Although we can apply quotas for applications and quotas for each individual user, if the number of applications or users increases suddenly that might also cause our target services to be loaded beyond their capacity. For example, assume the maximum capacity our target services can tolerate is 1500 transactions per second. And if we allow each application to consume 100 transactions per second, we can tolerate a maximum of 15 applications at full capacity. Imagine a situation where there are more than 15 applications each consuming at least 100 transactions per second. In this situation each application would be well within its quota but still go beyond our maximum tolerance limit of 1500. The same situation can occur when an application experiences a spike in usage from its users. See figure 5.6 for an illustration.

More than 15 client applications each operating within their quota of 100 requests per second.

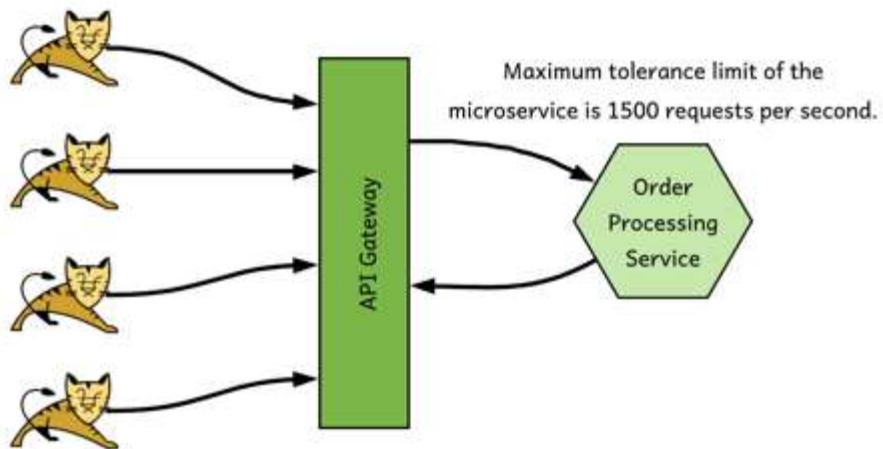


Figure 5.6 Figure illustrating a situation where there are more than 15 client applications operating within their quota of 100 requests per second. The combined load causes the maximum tolerable limit to exceed at the microservice.

These types of spikes can usually occur on special events. For example, if you run an ecommerce application, probably you will experience similar occurrences on special events such as thanksgiving, super bowl, etc. In most cases when the spike is predictable beforehand, you can scale up the target service layer for the relevant time period only. We have worked with a customer running a ride sharing business who rose in popularity and success in a very short time period, but unfortunately was unable to practically scale the target services layer due to design limitations. Their unexpected growth caused them to receive an unusually high amount of traffic during Friday evenings and Saturdays. This caused their target services layer to fail resulting in a total loss of business when the demand was at its most.

To handle these types of unprecedented surges we need to have a maximum threshold limit for our target services. Having such a policy prevents potential crashes on sudden surges resulting in a total loss of availability. But it also results in some users and applications to be denied of service even though they operate within the allowed quota. The tradeoff here is that we operate at maximum possible capacity servicing everyone possible instead of facing a total system failure resulting in service unavailability for everybody. The particular customer we mentioned above who was having a ride sharing business used this solution until they could rectify the design limitations that caused their services to be unscalable beyond a certain point. Figure 5.7 illustrates this scenario.

More than 15 client applications each operating within their quota of 100 requests per second.

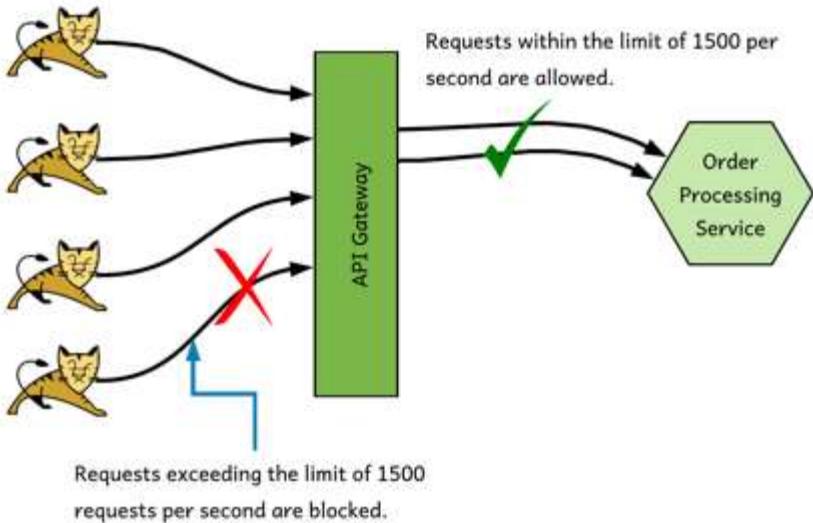


Figure 5.7 The gateway ensures it does not allow more than the maximum tolerable limit of the microservice to pass through.

5.1.5 Operation level throttling

An API hosted on the API gateway, can provide a level abstraction over one or more microservices. In other words, an API could have a one-to-many relationship with microservices. As we discussed in chapter 3 it sometimes makes sense to expose microservices as operations of a single API. When we do this we sometimes require applying throttling per each operation of the API instead of applying for the entire API as a whole. For example, in our Order Processing microservice, we may have one microservice, which performs read operations on our orders and another microservice, which performs write operations. If these two microservices are exposed via the same API as two operations, one as a `GET /orders` and the other as a `POST /orders` operation, you would most likely want to allow different quotas for each.

We were once working with a customer whose business was in the ticketing industry. They had APIs with operations that allowed their consumers to search for tickets as well as purchase APIs. They had an application that allowed their users to perform these search and purchase operations. You had to be logged into the application (authenticated) in order to make a purchase, but you could perform search operations without logging in (anonymously). Their usage pattern was such that they served a huge number of search operations compared to the number of purchase operations. They therefore had limited their quotas based on the

operations they served to make sure their usage pattern remains consistent with their observations and expectations. While having more purchases can be a good thing in terms of revenue, they wanted to stay safe. Any abnormality had to be first prevented and then allowed based on validation of legitimacy. Diagram 5.8 illustrates the scenario described above.

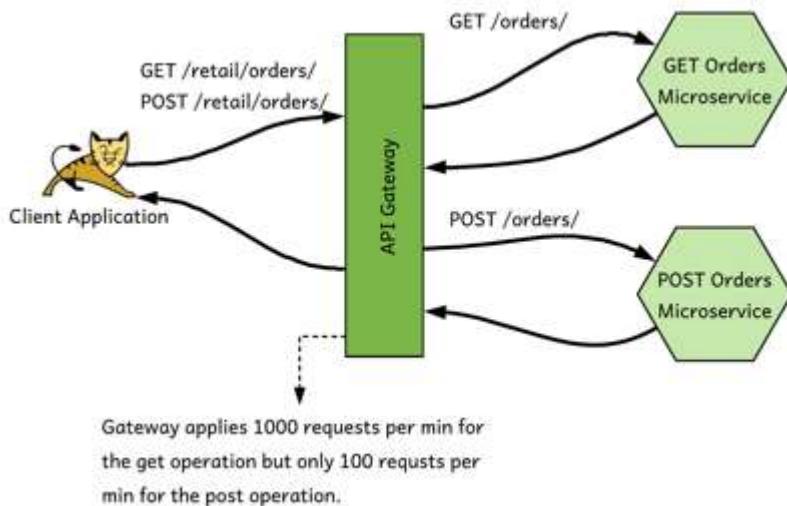


Figure 5.8 Illustrating how the gateway applies different level of throttling to different operations of the same API.

5.1.6 Throttling the OAuth 2.0 token and authorize endpoints

We have so far been discussing about throttling business operations of APIs. But, how about the login operation or the API (or the endpoint) that we use to authenticate users? Throttling this endpoint too is very important because a denial of service attack on the authentication endpoint along could make it impossible for a legitimate user to obtain tokens to access APIs.

However, throttling these APIs are tricky. Unlike other APIs, these APIs are invoked at a pre-authentication phase. Whoever is accessing these APIs are requesting to be logged in and therefore not yet authenticated. Whatever credentials the users present at this point may or may not be valid. We can only assess their legitimacy after performing the necessary validations. If this is an attacker, the validations are the exact points the attacker wants to exhaust, resulting in denial of services for all others. Therefore, things like application-based quotas or fair usage quotas for users cannot be applied in practice for this use case. This is because any attempt we do to identify the application or user at this point is in vain since we have no clue as to their legitimacy.

To uniquely identify the originator of these requests we therefore need to drop down to IP addresses. We need to identify the source IP address from where these requests are originating from and apply quotas for the unique IP addresses. To identify the source IP address from

which a request originated from, we can make use of the X-Forwarded-For header¹. Doing this at large scale and for Internet facing applications, goes beyond the scope of an API gateway. In such case we use a Web Application Firewall (WAF), which runs before the API gateway and intercepts all the requests. Imperva, Akamai, Cloudflare, AWS WAF are some popular WAF solution providers that also provide Denial of Service (DOS) and DDoS (Distributed DoS) prevention.

5.1.7 Privilege based throttling

Enforcing throttling based on different user privilege levels is another common use case we see in the industry. A user with a higher privilege may require to be allowed a larger quota of requests compared to a user with a lesser amount of privileges. Most of us would have come across scenarios where you get a limited quota of a particular service when you use a free account and larger quota of the service for paid accounts.

As we discussed in chapter 4 in detail, we can use OpenID Connect claims to determine the privilege level of the corresponding user and apply the relevant throttling limit to a client application. The API gateway intercepts the request originating from the client application and determines the privilege level of the end-user. It can then enforce throttling limits on the client based on the end-user's privilege level. The gateway would either receive the claims from the client application itself in the form of a JWT (a self-contained access token) or by querying information regarding the access token using the /userinfo endpoint of the authorization server. If you are new to OpenID Connect, please check appendix D and chapter 4. Figure 5.9 below illustrates this workflow.

¹ The X-Forwarded-For (XFF) HTTP header field is a common method for identifying the originating IP address of a client connecting to a web server through an HTTP proxy or load balancer

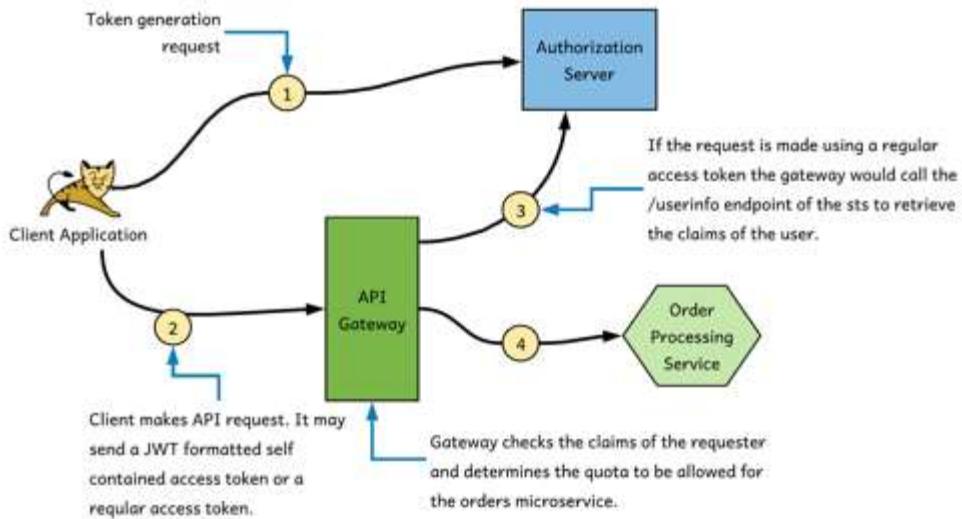


Figure 5.9 The workflow for applying privilege-based throttling for requests. The gateway determines the quota by inspecting the claims of the requester.

5.2 Monitoring and analytics with Prometheus and Grafana

In this section, we go into the details of monitoring a microservices deployment. The modern term for **monitoring and analytics** is known as **observability**. We discuss the importance of monitoring a microservices deployment and why it is critical to do so compared to monolithic applications in appendix F. In this section we discuss a few technologies we can use for the purpose of monitoring microservices, mostly Prometheus and Grafana.

Prometheus is one of popular open source monitoring tool for microservices. It helps us to keep track of system metrics over a given time period and can be used to determine the health of a software system. Metrics include things like memory usage, CPU consumption and a lot more. Grafana is an open source data visualization tool. It can help you build dashboards to visualize the metrics being provided by Prometheus or any other data source. At this time in writing Grafana is the most popular data-visualizing tool in the market. Although Prometheus has its own visualization capabilities, the features supported by Grafana are far more superior. Its visualization effects are much more appealing than Prometheus itself is.

5.2.1 Monitoring the Order Processing microservice

In this section we discuss what it means to monitor the Order Processing microservice using Prometheus and Grafana. To run this exercise, you need to have Docker (<https://www.docker.com/>) installed on your computer. We discuss Docker in detail in appendix A and therefore we don't go through Docker's basics in this section. Here we are going to run the Order Processing microservice, which is developed with Spring Boot, which exposes some system metrics over an endpoint (URL). We will then setup Prometheus to read

these metrics from this endpoint. Once that is done we will setup Grafana to visualize these metrics by getting the relevant information from Prometheus.

First check out the samples of this section from chapter05/sample02 of <https://github.com/microservices-security-in-action/samples>. Navigate to the chapter05/sample02 directory using your command line client tool and execute the following command to build the source code of the Order Processing microservice.

```
\> mvn clean install
```

Once the microservice is successfully built you can start the service by executing the command below.

```
\> mvn spring-boot:run
```

If you see a message as below that would mean the Order Processing microservice is up and running successfully.

```
Started OrderProcessingApp in 3.206 seconds
```

For Prometheus to be able to monitor the Order Processing microservice, the microservice needs to expose its metrics via a publicly accessible endpoint. Prometheus reads various metrics from the Order Processing microservice through this endpoint. This process is called **scraping** and we will discuss about it in detail later. To take a look at how these metrics are exposed, you can open up your web browser and access the following URL: <http://localhost:8080/actuator/prometheus>. You should see an output, which looks similar to listing 5.2.

Listing 5.2. The output from the actuator/prometheus endpoint

```
# TYPE system_load_average_1m gauge
system_load_average_1m 2.654296875
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="nonheap",id="Code Cache",} 2.5165824E8
jvm_memory_max_bytes{area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space",} 1.073741824E9
jvm_memory_max_bytes{area="heap",id="PS Eden Space",} 6.2652416E8
jvm_memory_max_bytes{area="heap",id="PS Survivor Space",} 4.456448E7
jvm_memory_max_bytes{area="heap",id="PS Old Gen",} 1.431830528E9
# HELP process_files_max_files The maximum file descriptor count
# TYPE process_files_max_files gauge
process_files_max_files 10240.0
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.552603801488E9
# HELP system_cpu_usage The "recent cpu usage" for the whole system
# TYPE system_cpu_usage gauge
system_cpu_usage 0.0
```

You can observe and figure out the type of metrics being exposed in listing 5.2. For example, the metric `jvm_memory_max_bytes` indicates the amount of memory being consumed by the

JVM (Java Virtual Machine). The metric process_start_time_seconds provides the time at which the process started, likewise. The following dependencies need to be added to the Spring Boot project of the Order Processing microservice for it to be able to expose this information. The dependencies are defined in the pom.xml file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

To enable exposure of the metrics at runtime, the following properties, needs to be enabled as below on the src/resources/application.properties file.

```
management.endpoints.web.exposure.include=prometheus,info,health
management.endpoint.prometheus.enabled=true
```

Now that we have our Order Processing microservice exposing the required metrics, the next step is to configure and start Prometheus so that it can read the exposed metrics. In this particular example we have created a docker-compose script which first starts the Prometheus container and then the Grafana container. To run these containers, navigate to the monitoring directory using your command line client and execute the following command.

```
\> docker-compose -f docker-compose.yml up
```

If this is the first time you are starting these containers it might take a few minutes for startup since it downloads the container images from Docker Hub (<https://hub.docker.com/>) and copies to your local docker registry. These containers would startup in a matter of seconds in subsequent attempts. Once the containers have started successfully you should see two messages as below.

```
Starting prometheus ... done
Starting grafana ... done
```

To see Prometheus in action, open up a new tab on your web browser and navigate to <http://localhost:9090>. From the top menu you see click on the drop-down list named **Status** and then click on **Targets**. You should be able to see a target named **order-processing** with its state as **UP**. You should see the last scrape timestamp information as well. This means that Prometheus is able to read the metrics exposed by our spring boot service.

Next click on the **Graph** link from the top menu. This UI in Prometheus allows you to query various metrics if you know the name of them. To check how much memory the JVM consumes, enter the string jvm_memory_used_bytes in the provided text box and click on the **Execute** button. The **Graph** tab would give you a view of the memory consumed over a period of time and the **Console** tab would show you the exact values at that particular moment. The **Graph** view would look similar to figure 5.10 as below.



Figure 5.10 – Graph view of the `jvm_memory_used_bytes` metric as displayed in the Prometheus UI.

To understand how Prometheus scrapes the Order Processing microservice for information, you can open up the monitoring/prometheus/prometheus.yml file. This is the Prometheus configuration file. The scrape configurations shown in listing 5.3 help Prometheus find the Order Processing microservice and its metrics.

Listing 5.3. The Prometheus scrape configuration

```
scrape_configs:
- job_name: 'order-processing'
  scrape_interval: 10s      #A
  metrics_path: '/actuator/prometheus'    #B
  static_configs:
- targets: ['host.docker.internal:8080']    #C
  labels:
    application: 'springboot-app'
```

#A Defines the frequency at which metrics should be collected from each target

#B Defines the path under which our metrics are hosted

#C Specifies the hosts which requires to be monitored under this job

As you would have noticed from using the Prometheus UI, it requires us to know the parameters to watch out for if we were to use the default Prometheus UI for monitoring our microservices. In terms of having an overall view of the state of a microservices deployment this experience/process does not help us a lot. This is where Grafana comes into the picture and helps us build dashboards for an overall view. Let us take a look at how we can create a dashboard in Grafana for a better view.

By now our Grafana container is up and running. We can therefore use it directly. To do that open a new tab on your browser console and navigate to `http://localhost:3000`. Enter **admin** as the username and password as the password to login to Grafana.

Next, we need to install a dashboard on Grafana. To do that hover over the **Create** menu item on the left side menu panel (it should be the first item in the menu) and click on the **Import** link. In the page that appears, click on the **Upload .JSON File** button and choose to upload the `monitoring/grafana/dashboard.json` file.

In the form that appears next, go with the defaults for all the fields except for the **Prometheus** field where you are expected to select a data source for this dashboard. Select **Prometheus** as the data source for its value as well and proceed to import this dashboard.

Next, under **Dashboards** of the left menu pane click on the **Manage** link. You should see a dashboard named as **JVM (Micrometer)**. Once you click on this dashboard you should see a number of widgets being loaded on to the UI. They are categorized into different sections such as **Quick Facts**, **JVM Memory**, etc. At first time load it might take a short while for the widgets to load. After they are loaded you should be seeing something similar to figure 5.11 below.

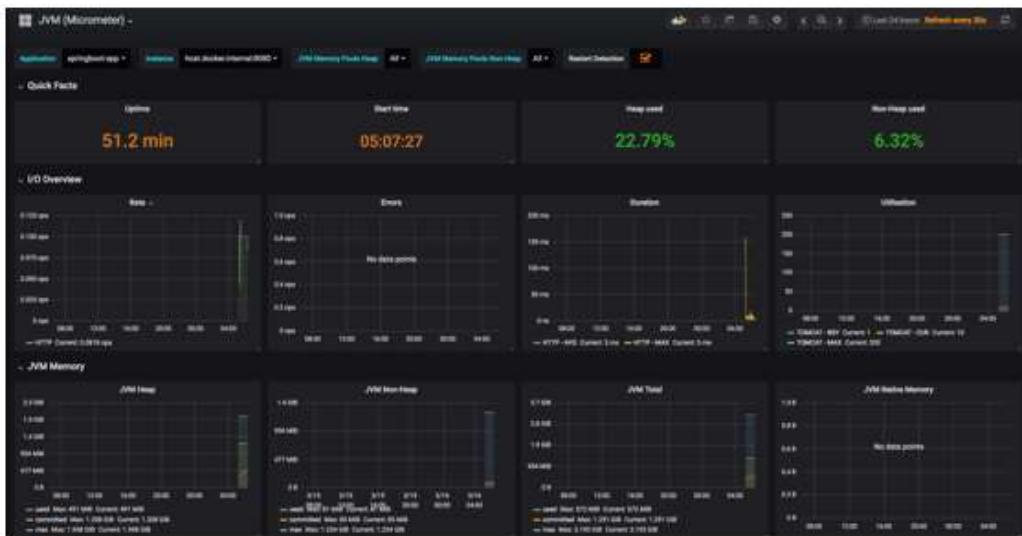


Figure 5.11 – The Grafana dashboard for our microservice. The metrics exposed by our microservice are scraped by Prometheus periodically. Grafana queries Prometheus using PromQL and visualizes the metrics.

As you may observe from the above, Grafana gives you a much more user-friendly view of the metrics exposed by the Order Processing microservice. To understand how Grafana queries data from Prometheus, you would need to take a look at the `monitoring/grafana/provisioning/datasources/datasource.yml` file. This file contains the Prometheus URL so that Grafana can connect to it and query its data. The `dashboard.json` file located in the `monitoring/grafana` directory defines, in JSON

format the type of metrics to be visualized under each widget. For example, take a look at the following JSON, which visualizes the **Uptime** panel on this dashboard.

```
"targets": [
{
  "expr": "process_uptime_seconds{application=\"$application\", instance=\"$instance\"}",
    "format": "time_series",
    "intervalFactor": 2,
    "legendFormat": "",
    "metric": "",
    "refId": "A",
    "step": 14400
  }
],
"thresholds": "",
"title": "Uptime",
. . . . .
```

We now have a bit of an experience on how to expose metrics from a microservice for Prometheus to scrape from and how to use Grafana to visualize these metrics.

5.2.2 Behind the scenes of using Prometheus for monitoring

Prometheus is an open source system monitoring and alerting tool. It is a standalone open source project maintained by the community itself. It is part of the Cloud Native Computing Foundation (CNCF) and the second hosted project in CNCF. As of this writing only 6 projects have graduated in CNCF and Prometheus is one of them. It is also the most popular open source monitoring tool available as of this writing. When using Prometheus to monitor a microservices deployment it is important to understand a few things regarding how Prometheus works.

SCRAPING DATA FROM MICROSERVICES TO MONITOR

Prometheus pulls metrics data from microservices on a periodic time interval. This is known as scraping. Each microservice needs to have an exposed endpoint, which contains details about the various metrics we need to monitor. The Prometheus server connects to these endpoints periodically and pulls down the information it needs for its monitoring purposes. Prometheus also has a push-gateway for supporting short-lived processes. Processes that may not live long enough for Prometheus to scrape can push their metrics to a push gateway before dying off. The push gateway acts as a metrics cache for the processes that no longer exists.

WHAT IS TIME SERIES DATA?

Prometheus stores metrics in a time-series database at millisecond precision. A time-series database contains a recording of various metrics against the time at which it was recorded. This data is stored for a period of time and usually presented in line graphs against time.

DEFINING A METRIC IN PROMETHEUS

A metric in Prometheus is an immutable block of data identified using both the metric name and labels. A metric is stored against its timestamp. Given a metric name and labels, time series are identified using the following notion.

```
<metric_name>=<label_name>=<label_value>, . . . }
```

For example, a metric used for getting the total number of HTTP requests would look like the following.

```
http_requests_total={method="POST", path="/menu", type="JSON"}
```

Figure 5.12 below illustrates the architecture of Prometheus.

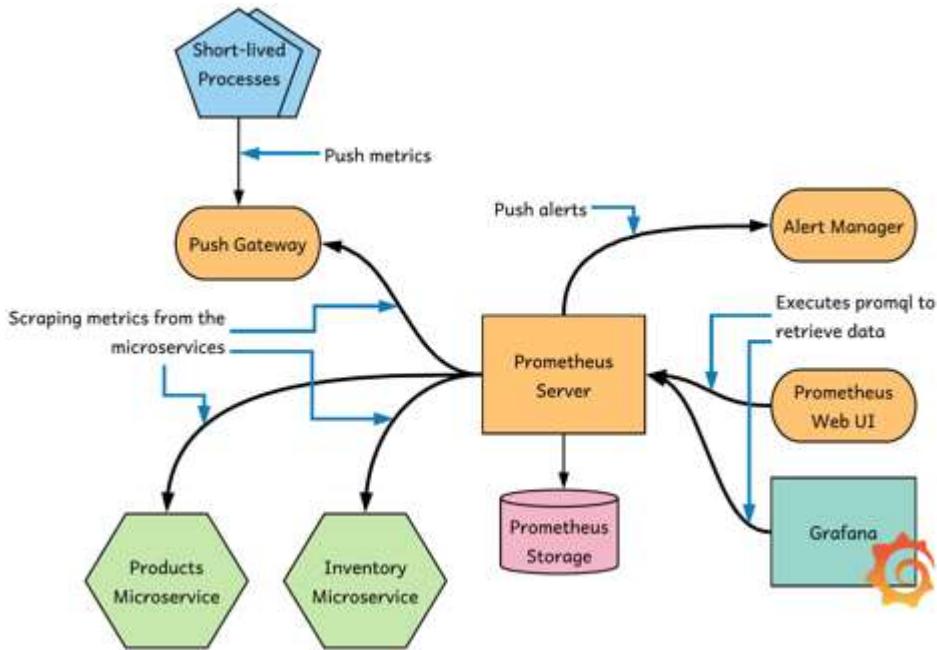


Figure 5.12 – Prometheus server scraping the microservices and push gateway for metrics. It then uses these metrics to trigger alerts. The Prometheus web UI and Grafana uses PromQL to retrieve data from Prometheus for visualization.

As you can see from figure 5.12 each microservice needs to expose an endpoint for Prometheus to scrape out metrics information from. When exposing this endpoint, we need to ensure that this endpoint is secured using TLS so that the information passed on the wire is kept safe from intruders and use an authentication mechanism such as OAuth 2.0 or Basic

Authentication. Prometheus has the capability access this endpoint using a Basic Authentication header or using a bearer token.

5.3 Enforcing access control policies at the API gateway with Open Policy Agent (OPA)

In this section, we look at controlling access to the Order Processing microservice using Open Policy Agent (OPA), at the API gateway. The API gateway here, is acting as a policy enforcement point. OPA is a lightweight general-purpose policy engine that has no dependency on microservices. You can use OPA to define fine-grained access control policies and enforce those policies at different places in a microservices deployment. In chapter 2 (under the section, service-level authorization with OAuth 2.0 scopes), we looked at how we can use OAuth scopes to control access to microservices. There we enforced OAuth scope based access control at the service-level by modifying the service code, which is not a good practice. Access control policies evolve as the business requirements change – so, every time we have to change our access control policies, changing the microservice code is not a good practice. OPA helps you externalizing access control policies and enforce them at any point in the request or response path. Figure 5.13 illustrates the sequence of events that happens when an API gateway intercepts client requests to apply authorization policies using OPA. OPA engine runs as a different process, outside the API gateway – and API gateway connects to OPA engine over HTTP. Please check appendix J for more details on OPA.

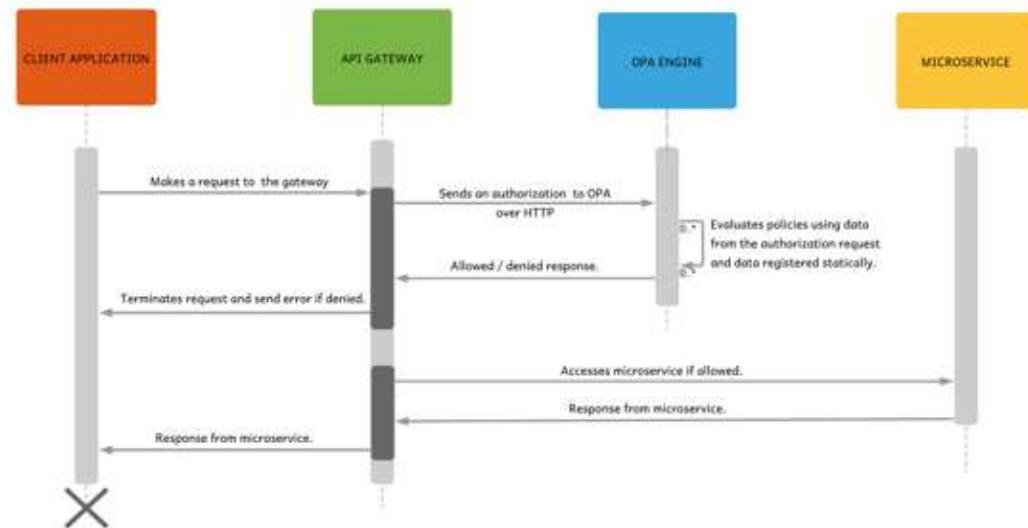


Figure 5.13 – Sequence of events that happens when an API gateway intercepts client requests to apply authorization policies using OPA.

5.3.1 Running Open Policy Agent (OPA) as a Docker container

In this section we discuss how to start OPA as a Docker container. That is the most straightforward and easiest way to get started with. As in the sample related to Prometheus, we need Docker installed and running on your machine to try this out along with the other perquisites mentioned in chapter 2 for running samples in this book in general.

First check out the samples of this section from `chapter05/sample03` of <https://github.com/microservices-security-in-action/samples>. As the first step of executing this sample we need to start the OPA Docker container (if you are new to Docker, please refer appendix A on Docker fundamentals). You can do this by using your command line client to execute the following command from within the `chapter05/sample03` directory. This script starts OPA server on port 8181 and binds it to the port 8181 of your local machine. So, please make sure that there is no other process running on port 8181. If you want to change the port, you can do it by editing the `run_opa.sh` file.

```
\> sh run_opa.sh
```

If your OPA container starts successfully you should see a message on your terminal window as follows. Note that if this is the first time you are running this command it might take a few minutes for the OPA Docker image to be downloaded from the Docker registry. Subsequent attempts would be much faster than the initial attempt.

```
{"addrs":[":8181"],"insecure_addr":"","level":"info","msg":"Initializing
server.", "time": "2019-11-04T01:03:09Z"}
```

5.3.2 Feeding OPA engine with data

Now that our OPA engine is up and running and can be accessed on port 8181, it is time to register the data required for executing policies. As you can observe, the listing 5.4 is a declaration of a collection of resource paths. These resources represent one or more resources corresponding to the Order Processing microservice. Each resource has an `id`, `path`, `method`, and a collection of `scopes` that are associated to the resource. The OPA's REST APIs allow registering these types of data sets on it. You can find the content of listing 5.4 in the file `chapter05/sample03/order_policy.json`. This data set is a collection of service paths (resources) where each resource declares the scope required for accessing it.

Listing 5.4. A set of resources in Order Processing microservice, defined as OPA data

```
[
{
  "id": "r1",      #A
  "path": "orders",    #B
  "method": "POST",    #C
  "scopes": ["create_order"]    #D
},
{
  "id": "r2",
  "path": "orders",
  "method": "GET",
  "scopes": ["retrieve_orders"]
```

```
},
{
  "id": "r3",
  "path": "orders/{order_id}",
  "method": "PUT",
  "scopes": ["update_order"]
}
]
```

#A An identifier for the resource path.
#B The resource path.
#C The HTTP method.
#D To do an HTTP POST to the order resource, you must have this scope.

You can register this data on OPA server by running the following cURL command from chapter05/sample03 directory.

```
\> curl -v -H "Content-Type: application/json" -X PUT --data-binary @order_policy.json
http://localhost:8181/v1/data/order_policy
```

You should see a response with the status code 204 if the request was successful. Please note that the `order_policy` element in the OPA endpoint, after the `data` element is important. OPA derives the package name for the data you pushed, using that. In this case, the data you pushed to the OPA server is registered under the `data.order_policy` package name. You can find more details on this on appendix J. To verify that your data has been successfully registered on the OPA engine you can execute the following cURL command. You should see the content of your resource definitions if the request is successful.

```
\> curl http://localhost:8181/v1/data/order_policy
```

Once the OPA engine has been initialized with the dataset required for our policies, the next step would be to implement and deploy access control policies on OPA.

5.3.3 Feeding OPA engine with access control policies

Let's see how to deploy authorization policies into the OPA server, which check whether a user/system accessing a resource, with an access token bears the scopes required for accessing that resource. We are using OPA policies to make authorization checks on the Order Processing microservice. OPA policies are written using a purpose-built language called Rego. It has rich support for traversing nested documents and transforming data using syntaxes similar to Python and JSONPath. Each policy written in Rego is a collection of rules that need to be applied on your microservice. Let's take a look at the policy defined in listing 5.5 that checks whether a token that is being used to access the Order Processing microservice bears the scopes that are required by the microservice.

Listing 5.5. OPA policy written in Rego

```
package authz.orders    #A
import data.order_policy as policies    #B
```

```

default allow = false      #C

allow {      #D
  policy = policies[_]    #E
  policy.method = input.method    #F
  policy.path = input.path
  policy.scopes[_] = input.scopes[_]
}

#A The package name of the policy.
#B Declares the set of statically registered data identified by order_policy, as in listing 5.4
#C All the requests by default are disallowed. If this is not set and no allowed rules matched, then OPA will return
     undefined decision
#D Declares the conditions to allow access to the resource.
#E Iterate over values in the policies array
#F For an element in the policies array check whether the value of the method parameter in the input, matches with the
     method element of policy

```

In listing 5.5, the package declaration is an identifier for the policy. If you want to evaluate this policy against certain input data, you need to make an HTTP POST request to the `http://localhost:8181/v1/data/authz/orders` endpoint having the input data as a JSON payload. Here we refer to the policy in the URL by `/authz/orders`. This is exactly same as the package declaration of the policy, with just the period character (.) being replaced by forward slash (/). You can find the policy we define in listing 5.5 in the `sample03/orders.rego` file. We can register this policy in OPA by executing the following command from within the `chapter05/sample03` directory.

```
\> curl -v -X PUT --data-binary @orders.rego http://localhost:8181/v1/policies/orders
```

We can execute the command below to verify that our policy has been registered successfully. If successful you should get the response containing the content of your policy.

```
\> curl http://localhost:8181/v1/policies/orders
```

5.3.4 Evaluating OPA policies

Once we have the OPA policy engine running with data and policies, we can use its REST API to check whether a given entity is authorized to perform a certain action. To send a request to the OPA policy engine, first we need to create an OPA input document. An input document will let OPA know details of the resource being accessed and details of the user who is accessing it. Such inputs are provided to the policy so that the policy can compare that with the set of statically defined data to make its decision. These inputs are provided in JSON format from the microservice (or the API gateway) to the OPA engine at the time of serving a business API request. Listing 5.6 shows an example of an input document that contains information of a particular request that is being served by the Order Processing microservice.

Listing 5.6. OPA input document

```
{
  "input":{
```

```

    "path": "orders",
    "method": "GET",
    "scopes": ["retrieve_orders"]
}
}

```

The above input document tells OPA that the microservice is serving a request on the path `orders` for an HTTP GET method. And there's a scope named `retrieve_orders` that's associated with the user (or the token) accessing the Order Processing microservice. OPA will use this input data and the statically declared data to run the rules declared in its policies.

Let's query OPA using its REST API to check if a particular input results in a true or false evaluation. We first evaluate a true case by using the input defined in `sample03/input_true.json`. You can evaluate this by executing the command below from `chapter05/sample03` directory.

```
\> curl -X POST --data-binary @input_true.json http://localhost:8181/v1/data/authz/orders -v
```

This should give you an HTTP 200 OK response with the following response body. What it means is that the details we used in the `input_true.json` file does match one of the rules in the policy registered on OPA. Please note that, as we discussed before, the OPA endpoint is derived from the package name of the policy we want to evaluate, which is **authz.orders** (see listing 5.5).

```
{"result":{"allow":true}}
```

If you execute the same command using the `input_false.json` file you would see a 200 OK response with the following content. Which means that you do not have rights to access the given resource with the given scope.

```
{"result":{"allow":false}}
```

5.3.5 Next steps in using OPA

Let's discuss some of the limitations and next steps, with respect to the OPA use case we discussed in the sections above. You can find how to address these limitations in appendix J.

- The connection to the OPA server, for evaluating policies is not properly secured. There are multiple options to secure OPA endpoints, which we discuss in appendix J.
- The OPA server runs as a Docker container, and all the policies and data pushed to the OPA server using APIs will be gone, when you restart the server. Once again in appendix J, we discuss how to over come that.
- In our example, we only use cURL client to evaluate OPA policies, against a given request (or an input document). If you would like to engage OPA with Zuul API gateway, then you need to write a Zuul filter, which is similar to the ThrottlingFilter we had in listing 5.1. This filter has to intercept the requests, create an input document and then talk to the OPA endpoint to see whether the request is authorized or not.

5.4 Summary

- Quota based throttling policies for applications help to monetize APIs/microservices and also limit a given application from over consuming APIs/microservices.
- Fair usage policies need to be enforced on applications to ensure all users get a fair quota of requests.
- User privilege based throttling is useful to allow different quotas for users with different privilege levels.
- An API gateway can be used to apply throttling rules in a microservices deployment.
- Metrics, tracing, logging and visualization are the four main pillars in observability.
- Each pillar is equally important to observe our microservices end-to-end so that we minimize impact on our business in case of a failure.
- Prometheus is the most popular open source monitoring tool available as of date of this writing.
- Grafana helps to visualize the data being recorded by Prometheus.
- Open Policy Agent (OPA) helps controlling access to a microservice deployment.
- OPA Data, OPA Input Data and OPA Policies are used together to apply various access control rules.

6

Securing east/west traffic with certificates

This chapter covers

- Creating certificates and securing microservices with mutual Transport Layer Security (mTLS)
- The challenges in certificate management, trust bootstrap, and certificate revocation in a highly distributed system
- The solutions and workarounds to address the challenges in securing microservices with mTLS

In the chapters 3, 4 and 5, we discussed how to expose a microservice as an API via an API gateway securely and apply other quality of service features such as throttling and monitoring. That's all part of the edge security in a typical microservices deployment. Edge security deals with authenticating and authorizing the end-user, which is a system accessing a microservice on behalf of a human user or another system. When the security screening at the edge is done, the end-user context is passed across to the upstream microservices. In this chapter we discuss securing communications among microservices with mutual Transport Layer Security (mTLS). mTLS is the most popular option for securing communications among microservices.

6.1 Why use mTLS?

When you buy something from Amazon, for example, all your credit card information flows from your browser to Amazon's servers over Transport Layer Security (TLS), and no one in the middle can see what it is. When you log in to Facebook, your credentials flow from your browser to Facebook's servers over TLS; no one in the middle can intercept the communications and find out what they are. TLS protects communications between two parties

for confidentiality and integrity. Using TLS to secure data in transit has been a practice for several years. Due to increased cybersecurity threats, recently it has become mandatory practice in any business that seriously worries about data security. In fact, from July 2018 onward, the Google Chrome browser (version 68.0.0+) indicates that any website that doesn't support TLS is insecure (<https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html>). Apart from protecting data in transit for confidentiality and integrity, TLS helps a client application identify the server that it communicates with. In the Amazon example, the browser is the client application, and when it talks to Amazon over TLS, it knows what it talks to as a result of the security model and the infrastructure built around TLS.

If Amazon wants to expose its services over TLS, it must have a valid certificate that is trusted by all the client applications that want to communicate with it. A certificate represents the corresponding server's public key and binds it to a common name. Amazon's public certificate (see figure 6.1, which is described in the next section), for example, binds its public key to the www.amazon.com common name. The most important and most challenging part in TLS is how we build trust between a client and a server.

6.1.1 Building trust between a client and a server with a certificate authority

How do you build trust between Amazon and all the browsers (client applications) that want to access it? A third party that is known to all the client applications signs the certificates given to services such as Amazon. This third party is a certificate authority (CA). Anyone who wants to expose services over the web that are protected with TLS must get its certificate signed by a trusted CA. Few trusted CAs are available globally, and their public keys are embedded in all browsers. When a browser talks to Amazon over TLS, it can verify that Amazon's certificate is valid (not forged) by verifying its signature against the corresponding CA's public key that's embedded in the browser. The certificate also includes the hostname of Amazon (which is the common name) so that the browser knows it's communicating with the right server. Figure 6.1 shows the certificate issued to www.amazon.com by the DigiCert Global CA.

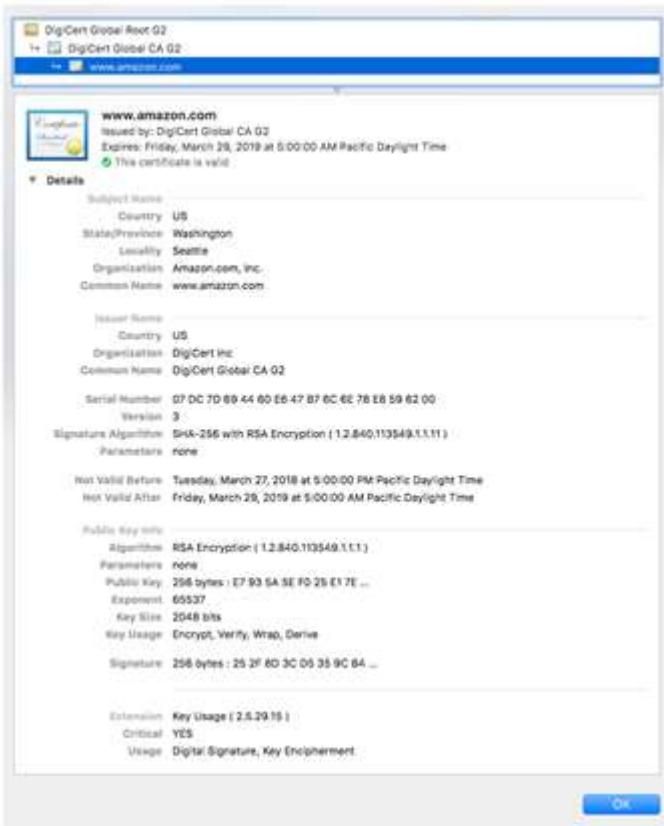


Figure 6.1 The certificate of www.amazon.com, issued it by the DigiCert Global CA. This certificate helps the clients talking to www.amazon.com to identify the server properly.

6.1.2 Mutual TLS helps the client and the server to identify each other

TLS itself is also known as *one-way TLS*, mostly because it helps the client identify the server it's talking to but not the other way around. Two-way TLS, or *mutual TLS*, fills this gap by helping client and server identify themselves to each other. Like the client knows what server it's talking to in one-way TLS, with mutual TLS (mTLS), the server knows the client it's talking to (see figure 6.2). To take part in a communication channel secured with mTLS, both the server and the client must have valid certificates, and each party must trust the issuer of the corresponding certificate. In other words, when mTLS is used to secure communications between two microservices, each microservice can legitimately identify who it talks to, in addition to achieving confidentiality and integrity of the data in transit between the two microservices.

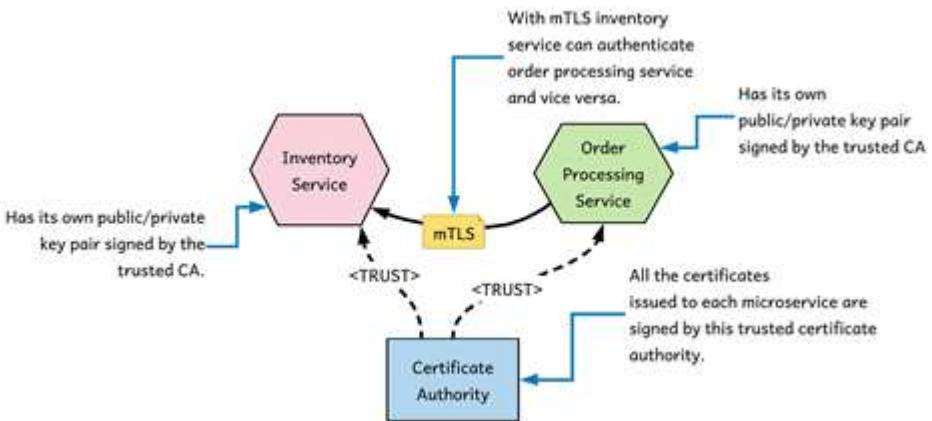


Figure 6.2 Mutual TLS between microservices lets them identify themselves. All the microservices in the deployment trust one certificate authority.

6.1.3 HTTPS is, HTTP over TLS

When you communicate with Amazon over TLS, the browser's address bar shows HTTPS instead of HTTP. HTTPS runs on TLS. In other words, HTTPS relies on TLS to provide security in the communication channel. TLS can be used by any application-layer protocol to make communications secure, not just HTTPS. A Java program can talk to a database by using Java Database Connectivity (JDBC), for example. To secure the JDBC connection between the Java program and the database, you can use TLS. Also, when an email client wants to talk to a server over a secured communication channel, it can use SMTP over TLS. There are many such examples.

6.2 Creating certificates to secure access to microservices

In this section, we explain how to create a public/private key pair for your microservice and get a trusted CA to sign it. In a typical microservices deployment, microservices aren't directly exposed to the public; mostly, the external clients interact with microservices via APIs. If your microservice endpoints aren't public, you don't need to have a public CA to sign the corresponding certificates. You can have your own CA, trusted by all the microservices in your deployment.

6.2.1 Creating a certificate authority (CA)

In a typical microservices deployment, you have your own CA, trusted by all your microservices. In appendix K, we show you how to create a CA by using OpenSSL (<https://www.openssl.org/>). OpenSSL is a commercial-grade toolkit and cryptographic library for TLS, available for multiple platforms. Before we start creating the CA using OpenSSL, let's

prepare a working environment. You need a key pair for your CA, the Order Processing microservice and the Inventory microservice. Create a directory structure as follows:

```
\> mkdir -p keys/ca
\> mkdir -p keys/orderprocessing
\> mkdir -p keys/inventory
```

To create CA's public and private key pair, follow the steps in section K.1 of the appendix K, and copy those keys (`ca_key.pem` and `ca_cert.pem`) to the `keys/ca` directory. In the next section, we discuss how to create a public/private key pair for the Order Processing and Inventory microservices and get the keys signed by the CA you created in this section. If you want to skip the detailed instructions in appendix K, and generate all the keys for the CA, Order Processing microservice and the Inventory microservice in one go, please check section 6.2.4.

6.2.2 Generating keys for the Order Processing microservice

To generate certificates for the Order Processing microservice, you can follow the steps in K.2 of the appendix K and at the end of the process, copy the generated keystore file (`app.jks`) to `keys/orderprocessing` directory, and rename it to `orderprocessing.jks`. This keystore file has the private and public key pair of the Order Processing microservice, and the CA created in section 6.2.1 signs the public key.

6.2.3 Generating keys for the Inventory microservice

You repeat the same process (described in section 6.2.2) for the Inventory microservice. You can follow the steps in K.1 of the appendix K and at the end of the process, copy the generated keystore file (`app.jks`) to `keys/inventory` directory, and rename the `app.jks` to `inventory.jks`. Figure 6.3 shows the setup of keystores for both Order Processing and Inventory microservices, with CA signed certificates. In section 6.3, we discuss how to use these two keystores to secure the communication between the two microservices over TLS.

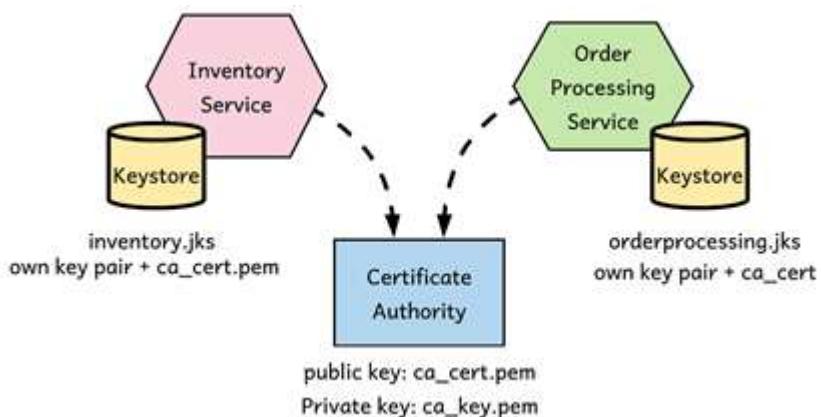


Figure 6.3 Keystore setup. Each microservice has its own private/public key pair, stored in a Java keystore file.

6.2.4 A single script to generate all the keys

In this section we introduce a single script to perform all the actions to create keys for the certification authority, Order Processing microservice, and Inventory microservice. If you have already followed the instructions in 6.2.1, 6.2.2 and 6.2.3, you can safely skip this section. The source code related to all the samples used in this chapter is available at <https://github.com/microservices-security-in-action/samples> in the GitHub repository's chapter06 directory.

First copy, **gen-key.sh** script from chapter06 directory to the `keys` directory that you created under section 6.2.1. Here we run OpenSSL in a Docker container. If you are new to Docker, please check appendix A, but you do not need to be thorough with Docker to follow rest of this section. To spin up the OpenSSL Docker container, run the following `docker run` command from the `keys` directory (listing 6.1).

Listing 6.1. Spins up OpenSSL in a Docker container

```
\> docker run -it -v $(pwd):/export prabath/openssl
#
```

The above `docker run` command starts OpenSSL in a Docker container, with a volume mount, which maps `keys` (or the current directory, which is indicated by `$(pwd)`) directory from the host file system to the `/export` directory of the container file system. This volume mount helps you to share part of the host file system with the container file system. When the OpenSSL container generates certificates, those are written to the `/export` directory of the container file system. Since we have a volume mount, everything inside the `/export` directory of the container file system is also accessible from the `keys` directory of the host file system.

When you run the command in listing 6.1 for the first time, it may take couple of minutes to execute and ends with a command prompt, where we can execute our script to create all the keys.

```
# sh /export/gen-key.sh
```

Now, if you look at the `keys` directory in the host file system, you will find the following set of files. If you want to understand, what happens underneath the script, please check appendix K.

- `ca_key.pem` and `ca_cert.pem` files under `keys/ca` directory.
- `orderprocessing.jks` file under `orderprocessing` directory.
- `inventory.jks` file under `inventory` directory.

6.3 Securing microservices with TLS

In this section, you develop two microservices, Order Processing and Inventory with Java, by using Spring Boot.¹ Then you enable TLS to secure communication between those two microservices.

¹ Spring Boot (<https://spring.io/projects/spring-boot>) is one of the most popular frameworks for developing microservices.

6.3.1 Running the Order Processing microservice over TLS

The Spring Boot sample of the Order Processing microservice is available in the chapter06/sample01 directory. Before you delve deep into the code, try to build, deploy, and run the Order Processing microservice. To build the sample, run the following Maven command from the chapter06/sample01 directory. When you run this command for the first time, it may take a considerable amount of time to complete. During the build process, Maven fetches all the binary dependencies that the Order Processing microservice needs. If those dependencies aren't available in the local Maven repo, it talks to remote Maven repositories and downloads them. That's how Maven works.

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

If everything goes well, a `BUILD SUCCESS` message is printed at the end. To run the microservice, use the following Maven command. Here, you use the Spring Boot Maven plugin (<https://docs.spring.io/spring-boot/docs/current/maven-plugin/usage.html>):

```
\> mvn spring-boot:run
INFO 21811 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
       6443 (http)
INFO 21811 --- [main] c.m.m.ch06.sample01.OrderProcessingApp : Started OrderProcessingApp
in 2.738 seconds (JVM running for 5.552)
```

If the service starts successfully, you find these two logs toward the end, saying that the Order Processing microservice is available on HTTP port 6443. Use the following curl command to test it. If everything goes well, the command returns a JSON response, which represents an order:

```
\> curl -k http://localhost:6443/orders/11
{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}
```

Now we'll show you how to enable TLS. First, press Ctrl-C to shut down the service. Then copy the `orderprocessing.jks` file, which you created earlier, from the `keys/orderprocessing` directory to the `chapter06/sample01` directory, where we have the source code of the Order Processing microservice. This keystore file contains the private/public key pair of the Order Processing microservice. Next, you need to edit the `chapter06/sample01/src/main/resources/application.properties` file and uncomment the following properties. In case you used different values for those parameters while generating the keystore, replace them appropriately.

```
server.ssl.key-store: orderprocessing.jks
server.ssl.key-store-password: manning123
server.ssl.keyAlias: orderprocessing
```

In addition to these properties, you'll find a `server.port` property in the same file. By default, this property is set to 6443. If you want to start the service on a different port, feel free to change the value.

You're all set. Rebuild the service and start it with the following commands:

```
\> mvn clean install
\> mvn spring-boot:run

INFO 21811 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
  6443 (https)
INFO 21811 --- [main] c.m.m.ch06.sample01.OrderProcessingApp    : Started OrderProcessingApp
  in 2.738 seconds (JVM running for 5.624)
```

If the service starts successfully, you'll find a log that says that the Order Processing microservice is available on HTTPS port 6443. Use the following curl command to test it over TLS:

```
\> curl -k https://localhost:6443/orders/11

{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}
```

You've got your Order Processing microservice running on TLS (or over HTTPS).

Behind the scenes of a TLS handshake

When you use curl to invoke a microservice secured with TLS, curl acts as the TLS client. As a browser talks to Amazon over HTTPS (or TLS), the microservice first shares its public certificate with the curl client before establishing a connection. This process happens during the TLS handshake. The TLS handshake happens between a TLS client and a TLS service, before they start communicating, shares certain properties required to establish the secure communication. When the client gets the public certificate of the TLS service, it checks whether a CA that it trusts has issued it. In other words, the public key of the CA that issued the certificate to the TLS service must be with the TLS client.

In the example in this section, the Order Processing microservice is protected with a certificate issued by your own CA, and by default, curl doesn't know about its public key. You have two options: Ask curl to accept any certificate it receives without validating whether the certificate is issued by a CA it trusts, or provide curl the public key of your CA. You chose the first option, using `-k` in the curl command to instruct curl to avoid trust validation. Ideally you should not do this in a production deployment.

6.3.2 Running the Inventory microservice over TLS

To enable TLS for the Inventory microservice, follow the same process you did before in section 6.3.1. The Spring Boot sample of the Inventory microservice is available in the `chapter06/sample02` directory. Before you build and run the service, copy the `inventory.jks` file, which you created before, from the `keys/inventory` to the `samples/chapter06/sample02` directory. This keystore file contains the private/public key pair of the Inventory microservice. Then, to enable TLS, uncomment the following properties in

the `chapter06/sample02/src/main/resources/application.properties` file. In case you used different values for those parameters while generating the keystore, replace them appropriately:

```
server.ssl.key-store: inventory.jks
server.ssl.key-store-password: manning123
server.ssl.keyAlias: inventory
```

In addition to these properties, you'll find the `server.port` property in the same file. By default, it's set to 8443. If you want to start the service on a different port, feel free to change the value. Now you're all set to build the project and start the microservice with the following Maven commands:

```
\> mvn clean install
\> mvn spring-boot:run

INFO 22276 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
 8443 (https)
INFO 22276 --- [main] c.m.m.ch06.sample02.InventoryApp           : Started InventoryApp in
 3.068 seconds (JVM running for 6.491)
```

If the service starts successfully, you see a log that says the Inventory microservice is available on HTTPS port 8443. Use the following curl command to test it over TLS:

```
\> curl -k -v -X PUT -H "Content-Type: application/json" -d
  '[{"code":"101","qty":1}, {"code":"103","qty":5}]' https://localhost:8443/inventory
```

If everything works, the item numbers from the request are printed on the terminal where the Inventory microservice is running. Now you have got both your Order Processing and Inventory microservices running on TLS (or over HTTPS). In the next section, you see how these two microservices talk to each other over TLS.

6.3.3 Securing communication between two microservices with TLS

You're a few steps from enabling TLS communication between the Order Processing and Inventory microservices. You need to make a few changes in both microservices, though, so shut them down for the moment (if they're already running). When the Order Processing microservice talks to the Inventory microservice over TLS, the Order Processing microservice is the TLS client. To establish a TLS connection, it has to trust the issuer of the server certificate that the Inventory microservice provides during the TLS handshake. In other words, the Order Processing microservice has to trust the CA, which you created earlier in this chapter (see figure 6.4).

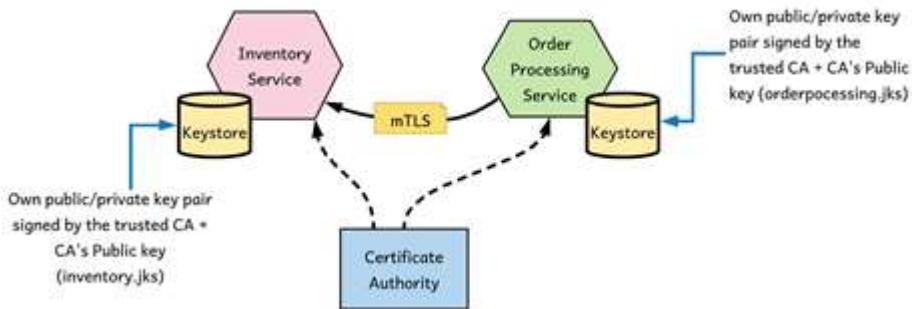


Figure 6.4 The Order Processing microservice talks to the Inventory microservice over TLS. Access to the Inventory microservice is protected with mTLS.

To trust a given CA, in Java (and also in Spring Boot), you need to explicitly specify a keystore in a system property (`javax.net.ssl.trustStore`) that carries the corresponding public key. You may recall that you imported the public key (`ca_cert.pem`) of your CA to both `orderprocessing.jks` and `inventory.jks`. Now you need to set the location of these keystores as a system property at the Order Processing microservice end. You don't need to set the same system property on the Inventory microservice side for the moment, because it doesn't do any calls to external microservices. Setting up the system property `javax.net.ssl.trustStore` is required only if a microservice acts as a TLS client. Uncomment the following code block (inside the static block) in `chapter06/sample01/src/main/java/com/manning/mss/ch06/sample01/OrderProcessingApp.java`:

```
// points to the path where orderprocessing.jks keystore is.
System.setProperty("javax.net.ssl.trustStore", path + File.separator +
    "orderprocessing.jks");
// password of the orderprocessing.jks keystore.
System.setProperty("javax.net.ssl.trustStorePassword", "manning123");
```

This change fixes one of your problems. The following code snippet from `chapter06/sample01/src/main/java/com/manning/mss/ch06/sample01/service/OrderProcessingService.java` shows how the Order Processing microservice talks to the Inventory microservice over TLS. When you POST an order to the Order Processing microservice, in the following code snippet, the Order Processing microservice talks to the Inventory microservice to update the item inventory. Here, you use the URL `https://localhost:8443/inventory`, which points to the Inventory microservice:

```
if (order != null) {
    RestTemplate restTemplate = new RestTemplate();
    URI uri = URI.create("https://localhost:8443/inventory");
    restTemplate.put(uri, order.getItems());

    order.setOrderId(UUID.randomUUID().toString());
    URI location = ServletUriComponentsBuilder
```

```

    .fromCurrentRequest().path("/{id}")
    .buildAndExpand(order.getOrderId()).toUri();
    return ResponseEntity.created(location).build();
}

```

What's the issue in this code snippet? You may recall that when you created the public certificate for the Inventory microservice, you used `iv.ecomm.com` as the value of the Common Name (CN) attribute. Any TLS client that talks to the Inventory microservice must use `iv.ecomm.com` as the hostname in the URL, not `localhost`. Otherwise, a hostname verification failure results. How do you fix this problem? The correct approach is to use the right hostname in the preceding code. But then you need to have a DNS setting pointing to the IP address of the server that runs the Inventory microservice, which is what you should do in production. For the time being, you can use a little trick. When you uncomment the following code snippet (inside the static block) in `chapter06/sample01/src/main/java/com/manning/mss/ch06/sample01/OrderProcessingApp.java`, the system automatically ignores the hostname verification:

```

HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
});

```

Now try service-to-service communication between the Order Processing and Inventory microservices over TLS. First, build and start the Order Processing microservice with the following Maven command from the `chapter06/sample01` directory, which you're already familiar with:

```
\> mvn clean install
\> mvn spring-boot:run
```

Next, start the Inventory microservice. Run the following Maven commands from the `chapter06/sample02` directory:

```
\> mvn clean install
\> mvn spring-boot:run
```

Now you've got both services running again. Use the following curl command to POST an order to the Order Processing microservice, which internally talks to the Inventory microservice over TLS to update the inventory:

```
\> curl -k -v -H "Content-Type: application/json" -d
  '{"customer_id":"101021","payment_method":{"card_type":"VISA","expiration":"01/22","name":"John Doe","billing_address":"201, 1st Street, San Jose, CA"},"items":[{"code":"101","qty":1},{"code":"103","qty":5}],"shipping_address":"201, 1st Street, San Jose, CA"}' https://localhost:6443/orders
```

If everything works, the item numbers from the request are printed on the terminal where the Inventory microservice is running.

6.4 Engaging mTLS

Now you have two microservices communicating with each other over TLS, but one-way TLS. Only the calling microservice knows what it communicates with, and the recipient has no way of identifying the client. This situation is where you need mTLS. In this section, you see how to protect the Inventory microservice with mTLS. When you have TLS set up between microservices, enabling mTLS is straightforward. First, shut down both the microservices, if they're already running. To enforce mTLS at the Inventory microservice end, uncomment the following property in the chapter06/sample02/src/main/resources/application.properties file:

```
server.ssl.client-auth = need
```

Setting this property to `need` isn't sufficient; you also need to identify which clients to trust. In this example, you're going to trust any client with a certificate signed by your CA. To do that, set the value of the system property `javax.net.ssl.trustStore` to a keystore that carries the public certificate of your trusted CA. You already have the public certificate of the trusted CA in the `inventory.jks` keystore, so all you have to do is set the system property that points to that keystore. Uncomment the following code block (inside the static block) in chapter06/sample02/src/main/java/com/manning/mss/ch06/sample02/InventoryApp.java.

```
// points to the path where inventory.jks keystore is.
System.setProperty("javax.net.ssl.trustStore", path + File.separator
    + "inventory.jks");
// password of inventory.jks keystore.
System.setProperty("javax.net.ssl.trustStorePassword", "manning123");
```

Next, build and spin up both the microservices to see how the interservice communication works. Build and start the Order Processing microservice, using the following Maven commands from the chapter06/sample01 directory:

```
\> mvn clean install
\> mvn spring-boot:run
```

Next, start the Inventory microservice. Run the following Maven commands from the chapter06/sample02 directory:

```
\> mvn clean install
\> mvn spring-boot:run
```

Now both services are running again. Use the following curl command to POST an order to the Order Processing microservice, which internally talks to the Inventory microservice over TLS to update the inventory. You expect this request to fail, as you enabled mTLS at the Inventory microservice end but didn't change the Order Processing microservice to authenticate with its private key:

```
\> curl -k -v -H "Content-Type: application/json" -d
  '{"customer_id":"101021","payment_method":{"card_type":"VISA","expiration":"01/22","name":"John Doe","billing_address":"201, 1st Street, San Jose,"}
```

```
CA"},"items":[{"code":"101","qty":1},{"code":"103","qty":5}],"shipping_address":"201,
1st Street, San Jose, CA"}' https://localhost:6443/orders
```

This request results in an error, and if you look at the terminal that runs the Order Processing microservice, you see the following error log:

```
javax.net.ssl.SSLHandshakeException: Received fatal alert: bad_certificate
```

The communication between the two microservices fails during the TLS handshake itself. To fix it, first take the Order Processing service down. Then uncomment the following code snippet (inside the static block) in chapter06/sample01/src/main/java/com/manning/mss/ch06/sample01/OrderProcessingApp.java. This code asks the system to use its private key from orderprocessing.jks to authenticate to the Inventory microservice:

```
// points to the path where orderprocessing.jks keystore is.
System.setProperty("javax.net.ssl.keyStore", path + File.separator +
    "orderprocessing.jks");
// password of orderprocessing.jks keystore.
System.setProperty("javax.net.ssl.keyStorePassword", "manning123");
```

Next, build and start the Order Processing microservice, using the following Maven command from the chapter06/sample01 directory:

```
\> mvn clean install
\> mvn spring-boot:run
```

Use the following curl command again to POST an order to the Order Processing microservice, and it should work this time:

```
\> curl -k -v -H "Content-Type: application/json" -d
    '{"customer_id":"101021","payment_method":{"card_type":"VISA","expiration":"01/22","name":
    "John Doe","billing_address":"201, 1st Street, San Jose,
    CA"},"items":[{"code":"101","qty":1},{"code":"103","qty":5}],"shipping_address":"201,
    1st Street, San Jose, CA"}' https://localhost:6443/orders
```

You have two microservices secured with TLS, and the communication between them is protected with mTLS.

6.5 Challenges in key management

Ask any DevOps person what the hardest part of the job is, and eight out of ten would say the key management. Key management is about how you manage keys in your microservices deployment. It involves four main areas: trust bootstrap and provisioning keys/certificates to workloads/microservices, key revocation, key rotation, and monitoring key use.

6.5.1 Key provisioning and trust bootstrap

In a typical microservices deployment, each microservice is provisioned with a key pair, as you did manually by copying Java keystore files to the Order Processing and Inventory microservices earlier in this chapter. Doing things manually won't work in a microservices

deployment, however; everything must be automated. Ideally, during the continuous integration/continuous delivery (CI/CD) pipeline, the keys should be generated and provisioned to the microservices. When the keys are provisioned to all the nodes/microservices, the next challenge is building trust between nodes. Why would a node trust a request initiated from another node?

One approach is to have a single CA for a given deployment and have each node in the deployment trust this CA. In other words, during the boot-up process of each node, you need to provision the public certificate of the CA to each microservice. This CA issues all node-specific keys. When one microservice talks to another one secured with mTLS, the recipient microservice validates the caller's certificate and verifies whether it's issued by the trusted CA (of the deployment); if so, it accepts the request. You followed this model in the Spring Boot examples earlier in this chapter.

TYPICAL KEY PROVISIONING PROCESS AT AN ENTERPRISE

The typical key provisioning mechanics that most enterprises use today don't deviate much from the approach you followed in this chapter while creating keys for the Order Processing and Inventory microservices. The developer who wants to secure a service with TLS first has to generate a public/private key pair, create a CSR, and submit the CSR to the team that maintains the corporate CA for approval. If everything looks good, the signed certificate is handed over to the developer who initiated the signing request. Then the developer deploys the certificate and the keys to the service. This process is painful in a microservices deployment with hundreds of services spinning up and down all the time.

KEY PROVISIONING AT NETFLIX

Netflix has thousands of microservices, and communication among those microservices is secured with mTLS. Netflix uses Lemur, an open-source certificate management framework, which acts as a broker (see figure 6.5) among the internal service deployment, CA, and management tools to automate the key provisioning process. During the process of continuous delivery, each microservice is injected with a set of credentials that are good enough to access the Lemur APIs. A tool called Metatron, which isn't yet open-source, does this credential bootstrap. While each microservice boots up, it talks to the Lemur API and gets a signed certificate for the corresponding public/private key pair. Lemur isn't a CA, but it knows how to integrate with a CA and generate a signed certificate. Microservices developers shouldn't worry about the certificate signing process, but about talking to the Lemur API. Figure 6.5 illustrates the key provisioning process.

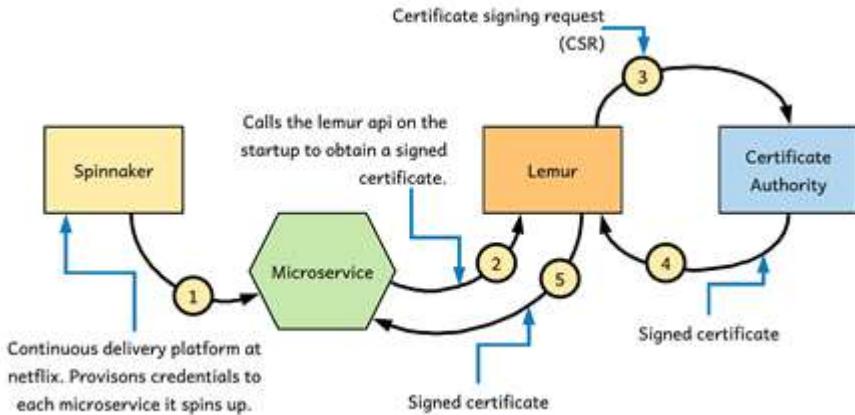


Figure 6.5 Key provisioning at Netflix. Each microservice at the startup talks to Lemur to get a signed certificate from the certificate authority in the domain.

GENERATING LONG-LIVED CREDENTIALS

In the key-provisioning model at Netflix, discussed in the preceding section, each microservice is provisioned with long-lived credentials that are used to connect to Lemur to get a signed certificate. The same long-lived credentials are used by each microservice when it has to refresh the keys to connect to Lemur. This method is a common solution to the trust bootstrap problem. Netflix uses a tool called Metatron to do the credential bootstrap. The internal details of Metatron aren't available yet for public access because the tool isn't open-source. In this section, however, we propose a scalable approach to generate long-lived credentials:

1. Protect the API of the certificate issuer (such as Lemur) so that anyone who wants to access it must present a valid key.
2. Build a handler to intercept the continuous delivery (CD) pipeline, which can inject long-lived credentials into the microservices.
3. Write the above handler in such a way that it generates the long-lived credentials as a JSON Web Token (JWT). This JWT will carry information about the node and will be signed by a key that's known to the certificate issuer. We discuss JWT in detail, in chapter 7.
4. At boot-up time, the microservice uses the injected long-lived credentials to talk to the certificate issuer's API and get a signed certificate. It can keep using the same long-lived credentials to rotate certificates.

SECURE PRODUCTION IDENTITY FRAMEWORK FOR EVERYONE (SPIFFE)

SPIFFE is an open standard that defines a way that a microservice (or a workload, in SPIFFE terminology) can establish an identity. SPIRE (SPIFFE Runtime Environment) is an open-source

reference implementation of SPIFFE. While helping establish an identity for each microservice in a given deployment, SPIFFE solves the trust bootstrap problem. We discuss SPIFFE in detail in appendix I.

6.5.2 Certificate revocation

Certificate revocation can happen for two main reasons: The corresponding private key is compromised, or the private key of the CA that signed the certificate is compromised. The latter situation may be rare, but in an internal CA deployment, anything is possible. A certificate can also be revoked for a third reason, which isn't as common in a private certificate deployment: If a CA finds the entity behind the signed certificate issued by it is no more represents the original entity at the time the certificate was issued or if it finds the details provided along with the certificate signing request is invalid, it can revoke the certificate. The challenge in certificate revocation is how to communicate the revocation decision to the interested parties. If the Amazon.com certificate is revoked, for example, that decision must be propagated to all browsers. Over time, multiple approaches have been suggested to overcome the challenges in certificate revocation. We go through some of these approaches in the following sections.

CERTIFICATE REVOCATION LISTS (CRLS)

Certificate Revocation List (CRL) was among one of the very first approaches suggested to overcome issues related to certificate revocation, defined in RFC 2459 (<https://www.ietf.org/rfc/rfc2459.txt>). Each CA publishes an endpoint where the TLS client applications can query and retrieve the latest revoked certificate list from that CA. As shown in figure 6.6, this endpoint is known as the CRL distribution point and is embedded in the certificate by the CA. According to RFC 5280 (<https://tools.ietf.org/html/rfc5280>), a CRL distribution point is a noncritical extension in a certificate. If a CA decides not to include it, it's up to the TLS client application to find the endpoint related to the corresponding CRL by some other means.



Figure 6.6 Amazon.com certificate embeds the corresponding CRL distribution points.

A CRL brings overhead to a TLS client application. Each time it validates a certificate, it has to talk to the CRL endpoint of the corresponding CA, retrieve the list of revoked certificates, and check whether the certificate in question is part of that list. The CRL can sometimes grow by megabytes. To avoid making frequent calls to the CA's CRL endpoint, client applications follow a workaround in which they cache the CRLs by CA. Every time they see the same certificate, they don't need to retrieve the corresponding CRL from the CA. This solution isn't good enough for highly security-concerned environments, however, because there's a possibility of making security decisions based on stale data. Also, CRLs create a coupling between the TLS client application and the CA. What would happen if the CRL endpoint of a given CA goes down? Should the TLS client application accept the certificates issued by that CA? This decision is tricky to make. With all these drawbacks and challenges, people started to move away from CRL-based certificate revocation.

ONLINE CERTIFICATE STATUS PROTOCOL (OCSP)

Unlike CRL, OCSP doesn't build one bulky list of all revoked certificates. Each time the TLS client application sees a certificate, it has to talk to the corresponding OCSP endpoint and check whether the certificate is revoked. As in CRL, the OCSP endpoint that corresponds to a given CA is also embedded in the certificate (see figure 6.7). Because the client application has to talk to the OCSP endpoint each time during the certificate validation process, the process creates a lot of traffic on the CA (or, to be precise, on the OCSP responder). Once again, as with CRL, to avoid frequent calls to the OCSP endpoint, some clients cache the revocation decisions. This solution, however, has the same issue as CRLs: Security decisions can be made based on stale data. Also, you still don't have a proper answer about what to do if the OCSP endpoint is down. Should the TLS client application accept the certificates issued by that CA? OCSP makes certificate revocation little better, but it doesn't fix all the challenges involved with CRLs. Google, in fact, decided not to support OCSP-based certificate validation in its Chrome browser; rather, it relies on frequent browser updates to share a list of revoked certificates.

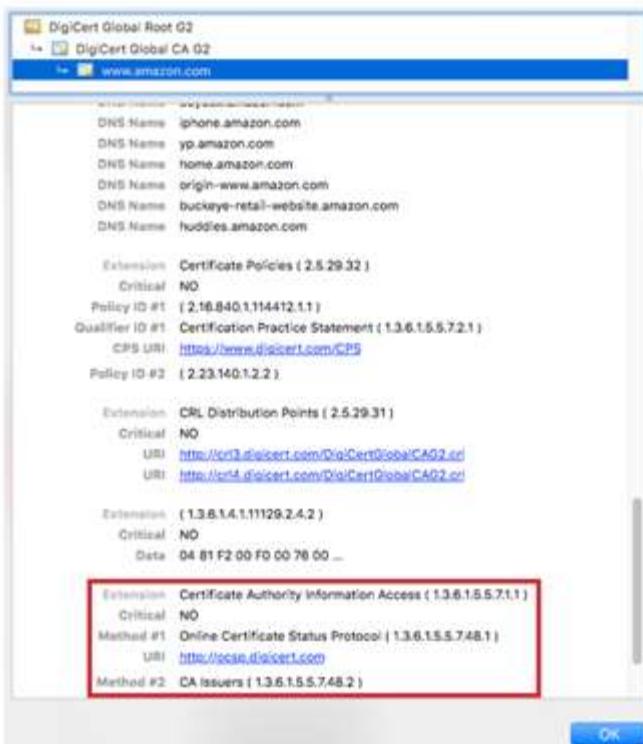


Figure 6.7 Amazon.com certificate embeds the OCSP endpoint.

Drawbacks in OCSP

The paper [Towards Short-Lived Certificates](http://bit.ly/2ZC9ISO) (<http://bit.ly/2ZC9ISO>) identifies four drawbacks in OCSP:

1. OCSP validation increases client-side latency because verifying a certificate is a blocking operation requiring a round trip to the OCSP responder to retrieve the revocation status (if no valid response is found in cache). A previous study indicates that 91.7 percent of OCSP lookups are costly, taking more than 100ms to complete, thereby delaying HTTPS session setup.
2. OCSP may provide real-time responses to revocation queries, but it's unclear whether the responses contain updated revocation information. Some OCSP responders may rely on cached CRLs on their backend. It was observed that DigiNotar's OCSP responder was returning good responses well after it was attacked.
3. Like CRLs, OCSP validations can be defeated in multiple ways, including traffic filtering and bogus responses forged by network attackers. Most important, revocation checks in browsers fail open. When they can't verify a certificate through OCSP, most browsers don't alert the user or change their UI; some don't check the revocation status at all. We must note that failing open is necessary, however, because there are legitimate situations in which the browser can't reach the OCSP responder.
4. OCSP also introduces a privacy risk: OCSP responders know which certificates are being verified by end users, so responders can in principle track which sites the user is visiting. OCSP stapling, which we discuss in the next section, is intended to mitigate this privacy risk but isn't often used.

ONLINE CERTIFICATE STATUS PROTOCOL (OCSP) STAPLING

OCSP stapling makes OCSP a little better. It takes the overhead of talking to the OCSP endpoint from the TLS client and hands it over to the server. This move is interesting and considerably reduces the traffic on the OCSP endpoint. Now the TLS server talks to the corresponding OCSP endpoint first, gets the signed response from the CA (or the OCSP responder), and attaches or staples the response to the certificate. The TLS client application by looking at the OCSP response attached to the corresponding certificate can check whether the certificate is revoked. This process may look a little tricky, because the owner of the certificate attaches the OCSP response to the certificate. If the owner decides to attach the same OCSP response throughout, even after the certificate is revoked, it can possibly fool the TLS client applications. In practice, however, this scenario is impossible. Each signed OCSP response from the CA (or the OCSP responder) has a timestamp, and if that timestamp isn't recent enough, the client must refuse to accept it.

ONLINE CERTIFICATE STATUS PROTOCOL (OCSP) STAPLING REQUIRED

Even with OCSP stapling, what would happen if no OCSP response is attached to the certificate? Should you reject the communication with the corresponding server or go ahead? This decision is hard to make. When OCSP stapling is made required, the server guarantees that the server certificate exchanged during the TLS handshake includes the OCSP response. If the client doesn't find the OCSP response attached to the certificate, it can refuse to communicate with the server.

SHORT-LIVED CERTIFICATES

The common consensus on certificate revocation is that it is a hard one to address. The approach suggested by short-lived certificates ignores certificate revocation, relying instead on expiration. But what is a good value for certificate expiration? Is it years or a couple of days? If the expiration time is too long, in case of a key compromise, all the systems that depend on the compromised key are at risk for a long time. The short-lived certificates suggest a short expiration, possibly a couple of days.

The concept of short-lived certificates is nothing new. It was initially discussed in 1998, but with microservices-based deployments, it has come back into the mainstream discussion. A research (<https://www.linshunhuang.com/papers/short-lived.pdf>) done at Carnegie Mellon University proposes using short-lived certificates to improve TLS performance. According to this proposal, CAs could configure the validity period of short-lived certificates to match the average validity lifetime of an OCSP response measured in the real world, which is four days. Such certificates expire quickly, and most important, the TLS client application rejects any communication afterward, treating them as insecure without the need for a revocation mechanism. Further, according to this proposal, when a website purchases a yearlong certificate, the CA's response is a URL that can be used to download on-demand short-lived certificates. The URL remains active for the year but issues certificates that are valid for only a few days.

NETFLIX AND SHORT-LIVED CERTIFICATES

In the Netflix microservices deployment, service-to-service communication is secured with mTLS and short-lived certificates. Netflix uses a layered approach to build a short-lived certificate deployment. Having a system identity or long-lived credentials residing in a Trusted Platform Module (TPM) or a Software Guard Extension (SGX) tightens security. During the boot-up process, access to the long-lived credentials is provisioned to each microservice. Then each microservice uses those credentials to get a short-lived certificate.

NOTE TPM is a hardware chip, which can securely store cryptographic keys, according to the TPM specification published by Trusted Computing Group (<https://trustedcomputinggroup.org/resource/tpm-library-specification/>). SGX by Intel allows applications to execute code and protect secrets in their own trusted execution environments. SGX is designed to protect secrets from malicious software.

Metatron, a tool developed by Netflix for credential management (which we discussed briefly in section 6.5.1), does the credential bootstrap. At this writing Metatron is in its beta version, and there are plans to open-source it in the future. When the initial credentials are provisioned to microservices, they talk to Netflix Lemur to get the short-lived certificates. Lemur (<https://github.com/Netflix/lemur>) is an open-source certificate manager developed by Netflix (see figure 6.8). Each microservice can refresh the short-lived certificates periodically, using its long-lived credentials. Each time a microservice gets a new certificate, the server environment must be updated with it. If you run your microservice on Spring Boot, for example, it should

know how to update all its transport senders and listeners with the updated certificate without restarting the server.

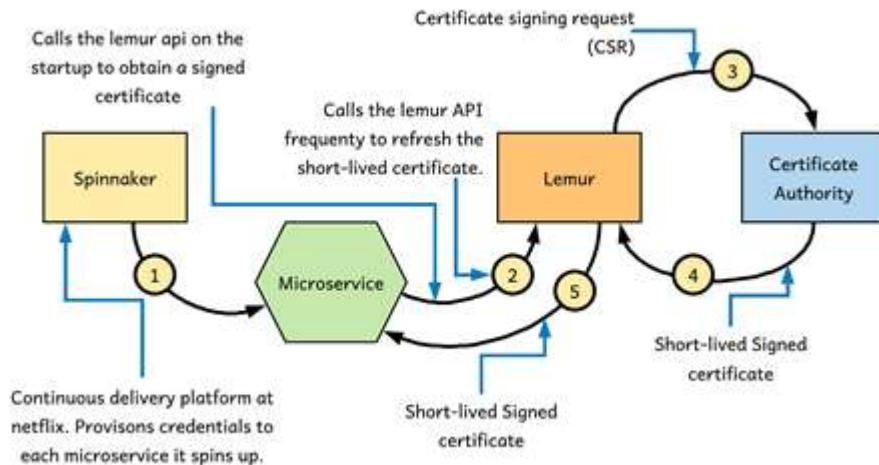


Figure 6.8 Netflix uses short-lived certificates with mTLS to secure service-to-service communication.

Why use long-lived credentials?

We assume in this section that long-lived credentials are well secured and hard to compromise. If that's the case, why are short-lived credentials necessary? Why not use much secured long-lived credentials themselves? The answer lies in performance. Long-lived credentials are well secured with a TPM or an SGX. Loading such long-lived credentials frequently is a costly operation. Short-lived credentials are kept in memory.

6.6 Key rotation

All the keys provisioned into microservices must be rotated before they expire. Not every enterprise worries about key rotation. Expiration times used to be higher, such as 5 to 10 years. You realize that you have to rotate certificates only when communication links start to fail due to the expired certificate. Some companies have key rotation policies that state that all the keys used in a deployment must be rotated every month or two. The short-lived certificate approach we discussed earlier in this chapter enforces certificate rotation in short intervals. Netflix, for example, in its microservice deployment rotates keys every four minutes. That interval may look crazy, but that's the level of security Netflix worries about. Then again, the key rotation policy differs from short-lived credentials to long-lived credentials.

The keys embedded in microservices can be short-lived and rotated much frequently, but your corporate CA's private key doesn't need to be rotated. The overhead of rotating the CA's private key is much higher. You need to make sure that every service that trusts the CA has the updated key. If it's hard to rotate some keys frequently, you should find better ways of

securing those keys. These better ways usually are expensive both financially and in terms of performance. Netflix uses a TPM or an SGX with tightened security to secure its long-lived credentials.

Key rotation is more challenging in a microservices deployment, with an increased number of services spinning on and off. Automation is the key to addressing this problem. Every microservices deployment requires an approach like the one Netflix uses with Lemur. SPIFFE is another approach, which we discuss in appendix I in detail.

6.7 Monitoring key usage

Observability is an essential ingredient of a typical microservices deployment. It's about how well you can infer the internal state of a system by looking at the external outputs. Monitoring, by contrast, is about tracking the state of a system. Unless you have a way to monitor the external outputs of a system, you'll never be able to infer the internal state of a system. Observability is a property of a system. Unless a microservice is observable, you won't be able to infer its internal state.

Logging, metrics, and tracing are the three pillars of observability. With logging, you can record any event happening in your system: a successful login event; a failed login event; success or failure in an access-control check; an event related to key provisioning, key rotation, or key revocation; and so on. Metrics indicate the direction of a system. Logging events helps you derive metrics. By tracking the time it takes to refresh keys in short intervals, for example, you can derive how much that process contributes to the average latency of the system. The latency of a system is reflected by the time interval between a request's entering and exiting a system. If the number of failed login attempts against a service is high or goes beyond a certain threshold, that service may be under attack, or a certificate may be expired or revoked.

Tracing is also derived from logs. Tracing worries about the order of events and the impact of one event on another. In a microservices deployment, if a request fails at the Inventory microservice, tracing helps you find out where the root cause is and what happened to the same request in the Order Processing and Delivery microservices. You can also trace which keys are being used between which services and identify the patterns in key use, which helps you identify anomalous behaviors and raise alerts.

Monitoring a microservices deployment is challenging, as many service-to-service interactions occur. We use tools like Zipkin, Prometheus, and Grafana in a microservices deployment to monitor key use.

6.8 Summary

- There are multiple options in securing communication between microservices: mutual TLS and JWTs.
- TLS protects communications between two parties for confidentiality and integrity. Using TLS to secure data in transit has been a practice for several years.
- Mutual Transport Layer Security (mTLS) is the most popular way of securing interservice

communication between microservices.

- TLS itself is also known as one-way TLS, mostly because it helps the client identify the server it's talking to but not the other way around. Two-way TLS, or mutual TLS, fills this gap by helping client and server identify themselves to each other.
- Key management in a microservices deployment is quite challenging and we need to worry about trust bootstrap and provisioning keys/certificates to workloads/microservices, key revocation, key rotation, and monitoring key use.
- Certificate revocation can happen for two main reasons: The corresponding private key is compromised, or the private key of the CA that signed the certificate is compromised.
- Certificate Revocation List (CRL) was among one of the very first approaches suggested to overcome issues related to certificate revocation, defined in RFC 2459.
- Unlike CRL, OCSP doesn't build one bulky list of all revoked certificates. Each time the TLS client application sees a certificate, it has to talk to the corresponding OCSP endpoint and check whether the certificate is revoked.
- OCSP stapling makes OCSP a little better. It takes the overhead of talking to the OCSP endpoint from the TLS client and hands it over to the server.
- The approach suggested by short-lived certificates ignores certificate revocation, relying instead on expiration.
- All the keys provisioned into microservices must be rotated before they expire.
- Observability is an essential ingredient of a typical microservices deployment. It's about how well you can infer the internal state of a system by looking at the external outputs. Monitoring, by contrast, is about tracking the state of a system.

7

Securing east/west traffic with JWT

This chapter covers

- The role of JSON Web Token (JWT) in securing service-to-service communication among microservices
- Using JWT to carry user context among microservices
- Using JWT for cross-domain authentication
- Using JWT for message level encryption and signature

In chapter 6, we discussed securing service-to-service communication in a microservices deployment with mutual Transport Layer Security (mTLS). mTLS is in fact the most popular option for authenticating one microservice to another. JSON Web Token (JWT), which provides a way to carry a set of claims or attributes from one party to another in a cryptographically secure way, too plays a key role in securing service-to-service communication in a microservice deployment. It can be used to carry the identity of the calling microservice, or the identity of the end user or the system that initiated the request. The JWT can also be used to propagate identity attributes between multiple trust domains. We explore in this chapter the role that JWT plays in securing service-to-service communication in a microservices deployment. If you are not familiar with JWT, we recommend you first go through appendix H. Appendix H provides a comprehensive overview of JWT.

7.1 Use cases for securing microservices with JWT

The JWT addresses two main concerns in a microservices security design: securing service-to-service communication and passing end-user context across microservices. As we discussed in

chapter 6, JWT isn't the most popular option for securing service-to-service communications; mTLS is. In this section we discuss, why you would pick JWT over mTLS to secure service-to-service communications and other use cases of JWT in a microservices deployment.

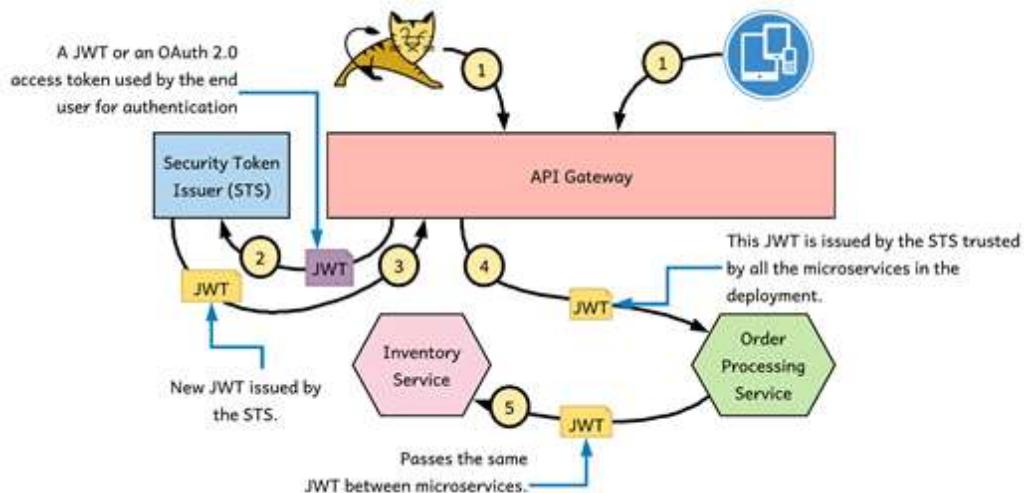


Figure 7.1 Propagating the end-user's identity in a JWT across microservices. All the microservices in the deployment trust the STS and the API gateway exchanges the JWTs it gets from client applications to new JWTs from this STS.

7.1.1 Sharing user context between microservices with a shared JWT

When the identity of the microservice isn't relevant, but the identity of the end user (system or human) is, you should prefer using JWT to mTLS. In this case, services themselves don't authenticate to one another. In every request, you need to carry the identity of the end user who initiates the message flow; if not, the recipient microservice will reject the request. The following list walks you through the numbered request flow shown in figure 7.1:

1. An end user initiates the request flow. This user can be human or a system.
2. As discussed earlier in the book under chapter 1 and chapter 3, the edge gateway authenticates the user. The edge gateway intercepts the request from the end user, extracts the token (which can be a JWT or an OAuth 2.0 access token), and then talks to the Security Token Service (STS) connected to it to validate. Then again, the token that the end user presents may be not issued by this STS; it can be from any other identity provider, which this STS trusts. The details related to the end-user authentication are discussed earlier in the book, under chapter 3. The STS should know how to validate the token presented to it in step 2.
3. After validating the token, the STS issues a new JWT signed by itself. This JWT includes the user details copied from the old JWT (from step 2). This step is an important one.

Whether or not the same STS issued first token (in step 2), this STS issues the new token. In other words, this STS signs the new JWT. When the edge gateway passes the new JWT to the upstream microservices, they need only trust this STS to check whether the token is valid. All the microservices in the deployment (within a single trust domain) trust a single STS.

4. The API gateway passes the new JWT issued by the STS in an HTTP header (Authorization Bearer) over TLS to the Order Processing microservice. The Order Processing microservice validates the signature of the JWT to make sure that it's issued by the trusted STS. Apart from the signature validation, the Order Processing microservice also does the audience validation, checking whether the value of the provided JWT's `aud` is known to itself (more details in appendix H). For the pattern discussed in this section to work, all the microservices in the same trust domain (that trust a single STS) must accept a JWT with a wildcard audience value such as `*.ecomm.com`.
5. When the Order Processing microservice talks to the Inventory microservice, it passes the JWT that it got from the API gateway. The Inventory microservice validates the signature of the JWT to make sure that it's issued by its trusted STS. Also, it checks whether the value of the `aud` attribute in the JWT is `*.ecomm.com`.

In this approach, JWT helps you achieve two things. For one, it helps you pass the end-user context across microservices in a manner that can't be forged. Because the claim set of the JWT is signed by the STS, no microservice can change its content without invalidating the signature. Also, JWT helps you secure service-to-service communications. One microservice can access another microservice only if it carries a valid JWT issued by the trusted STS. Any recipient microservice rejects any request without a valid JWT.

7.1.2 Sharing user context with a new JWT for each service to service interaction

The use case we discussed in this section is a slight variation of the one we discussed in section 7.1.1, but only the end user's identity is relevant—not the identity of the microservice. Instead of passing the same JWT across all the microservices and accepting the same audience value at each microservice, you generate a new JWT for each service interaction. This approach is much more secured than using a shared JWT. But there's no such thing as absolute security. Everything depends on your use cases and the level of trust you have in each microservices deployment.

Figure 7.2 illustrates how this pattern works. It's the same flow discussed in section 7.1.1 except for steps 4a and 4b. In step 4a, before the Order Processing microservice talks to the Inventory microservice, it talks to the STS and does a token exchange. It passes the JWT it got from the API gateway (issued under `op.ecomm.com` audience) and requests a new JWT to access the Inventory microservice. In step 4b, STS issues a new JWT under the audience `iv.ecomm.com`. Thereafter, the flow continues as in the preceding section.

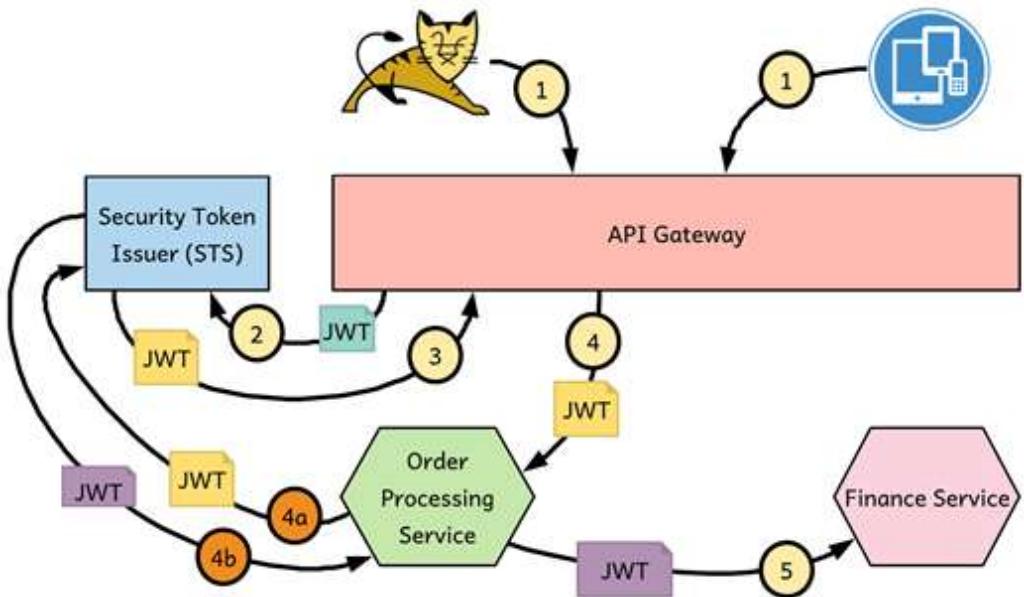


Figure 7.2 Propagating the end-user's identity in a JWT across microservices with token exchange.

Why do you need a new JWT with a new audience value when the Order Processing microservice talks to the Inventory microservice? Why it is more secure than sharing the same JWT coming from the API gateway across all the microservices in the deployment and accepting a single audience value? There are two valid reasons.

The first reason is, when you have one-to-one mapping between a microservice in your deployment and the audience value of the corresponding JWT issued by the STS, for a given JWT, you know exactly who the intended audience is. In step 4 of figure 7.2, for example, when the request is dispatched to the Order Processing microservice from the API gateway, it can make sure that the token goes to no other microservice but to the Order Processing.

The second reason is, if the Order Processing microservice tries to reuse the token given to it as is to access another service, such as the Finance microservice (which ideally, it shouldn't have access to), the request will fail because the audience value in the original JWT doesn't work with the Finance microservice, which has its own audience value. The only way that the Order Processing microservice can talk to the Finance microservice is to pass its current JWT to the STS and exchange it for a new JWT with an audience value accepted by the Finance microservice. Now you have more control at the STS, and the STS can decide whether to let the Order Processing microservice access the Finance microservice.

One would argue that, what's the point of doing an access control check at the STS, at the point of token exchange, while we can anyway do it at the edge of the microservice. Enforcing access control at the edge of a microservice is a common pattern, mostly with the Service

Mesh architecture, which we discuss in appendix C. In this case, since we do not worry about the identity of the microservices, the recipient microservice has no way to figure out who the calling microservice is, unless the STS embeds identity information about the calling microservice into the new JWT it created, at the point of token exchange. Even in that case, it is better to do coarse-grained access control checks at the STS, and push fine-grained access control checks to the edge of the microservice.

7.1.3 Sharing user context between microservices in different trust domains

The use case in this section is an extension of the token exchange use case discussed in section 7.1.3. As figure 7.3 shows, most of the steps are straightforward. There's no change from the preceding section up to step 6. (Steps 5 and 6 in figure 7.3 are equivalent to steps 4a and 4b in figure 7.2.) In step 7, the Order Processing microservice from the ecomm domain tries to access the Delivery microservice in the delivery domain via the delivery API gateway. The JWT carried in this request (step 7) is issued by the STS in the ecomm domain and has an audience value to match the Delivery microservice. In step 8, the API gateway of the Delivery domain talks to its own STS to validate the JWT. The validation passes only if the delivery STS trusts the ecomm STS. In other words, the corresponding public key of the signature in the JWT issued by the ecomm STS must be known to the delivery STS. If that is the case, in step 9, the delivery STS creates its own JWT and passes it over to the Delivery microservice via the API gateway. All the microservices in the delivery domain trust only the their domain's own STS.

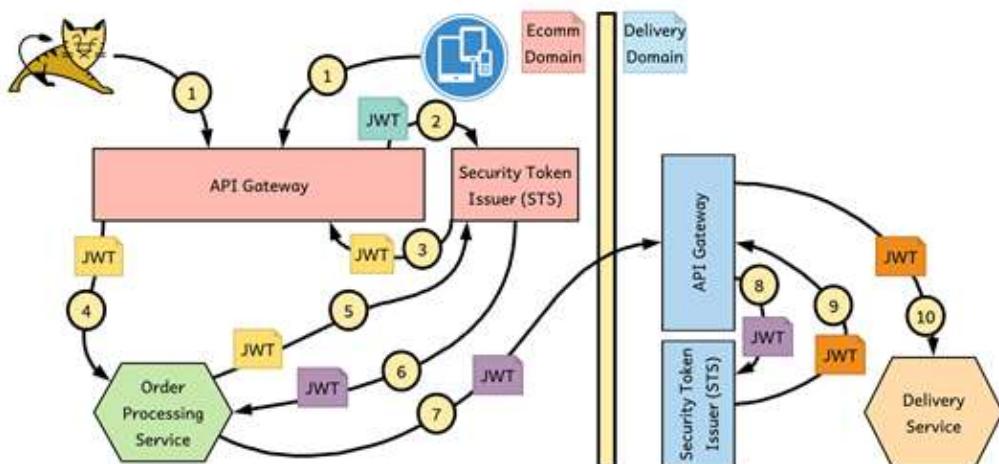


Figure 7.3 Cross-domain authentication and user context sharing among multiple trust domains. The STS in the delivery domain trusts the STS in the ecomm domain.

7.1.4 Self-issued JWTs

In the use cases discussed so far, you didn't worry about the identity of the microservice itself. Rather, you relied on a JWT issued by a trusted STS that carries the end user's identity. With self-issued JWTs (see figure 7.4), you worry about the identity of microservices when they talk to one another, as in mTLS (discussed in chapter 6). As in mTLS, even in this model, each microservice must have its own private/public key pair. Each microservice generates a JWT; signs it with its own private key; and passes it as an HTTP header (Authorization Bearer), along with the request, to the recipient microservice over TLS. Because the JWT in this case is a bearer token, the use of TLS is highly recommended (or in other words a must). The recipient microservice can identify the calling microservice after verifying the JWT signature by using the corresponding public key.

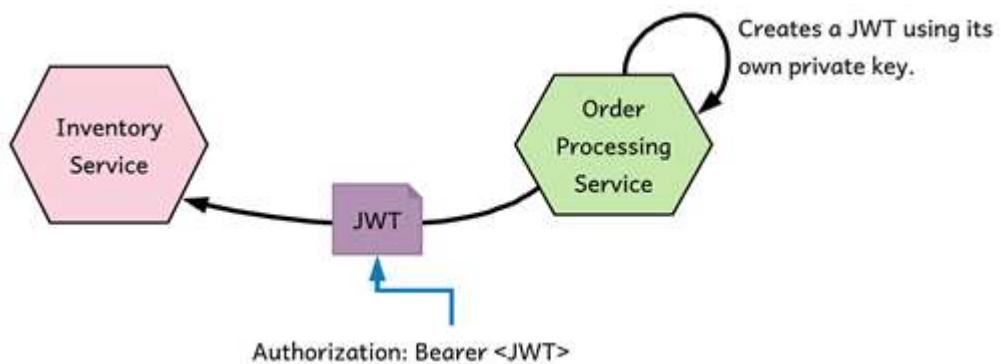


Figure 7.4 Self-issued JWT. The JWT is signed using the private key of the Order Processing microservice.

How does this process differ from mTLS? If what you're trying to achieve is only authentication between two microservices, neither method is superior. From the developer-overhead point of view, setting up mTLS is straightforward than using self-issued JWTs. Both use cases need to handle all the key management challenges discussed in chapter 6, along with service-to-service authentication. If you intend to share some data between two microservices, the self-issued JWT is much better than mTLS. If the Order Processing microservice wants to share the order ID with the Inventory microservice as a correlation handle, for example, it can embed it in the JWT. In case of mTLS, you need to pass it as an HTTP header.

What's the difference? mTLS provides confidentiality and integrity of the data in transit, but not nonrepudiation. Nonrepudiation cryptographically binds an action to the person who initiated it so that he or she can't deny it later. With mTLS alone, you can't achieve nonrepudiation. But when you use a self-issued JWT, all the data added to it is bound to the owner of the corresponding private key that's used to sign the message, which helps you achieve nonrepudiation. Even if you use a self-issued JWT, communication between the two microservices in most cases is over TLS (not mTLS), which protects the communication for

confidentiality and integrity. If you want to get rid of TLS, you can use a signed, encrypted JWT and still achieve those properties. But you'll rarely want to get rid of TLS.

NOTE A JWT is a bearer token. A bearer token is like cash. If someone steals 10 bucks from you, she can use it at any Starbucks to buy a cup of coffee, and no one will ask her to prove that she owns the 10 bucks. Anyone who steals a bearer token can use it with no issue, till the token expires. If you use JWT for authentication between microservices (or in other words, authenticate one microservice to another), you must secure the communication channel with TLS to minimize the risk of an intruder stealing the token. Also, make sure the JWT has a very short expiration. In that case, even someone steals the token; the impact of the stolen token is minimum.

7.1.5 Nested JWTs

The use case in this section is an extension of the use case discussed in the section 7.1.4. A *nested JWT* is a JWT that embeds another JWT (see figure 7.5). When you use a self-issued JWT to secure service-to-service communications between two microservices, for example, you can embed the JWT issued by the trusted STS, which carries the end user context, in the self-issued JWT itself and build a nested JWT. On the recipient side, it has to validate the signature of the nested JWT with the public key corresponding to the calling microservice, and validate the signature of the embedded JWT with the corresponding public key from the trusted STS. The nested JWT carries the identities of the end user and the calling microservice in a manner that can't be forged.

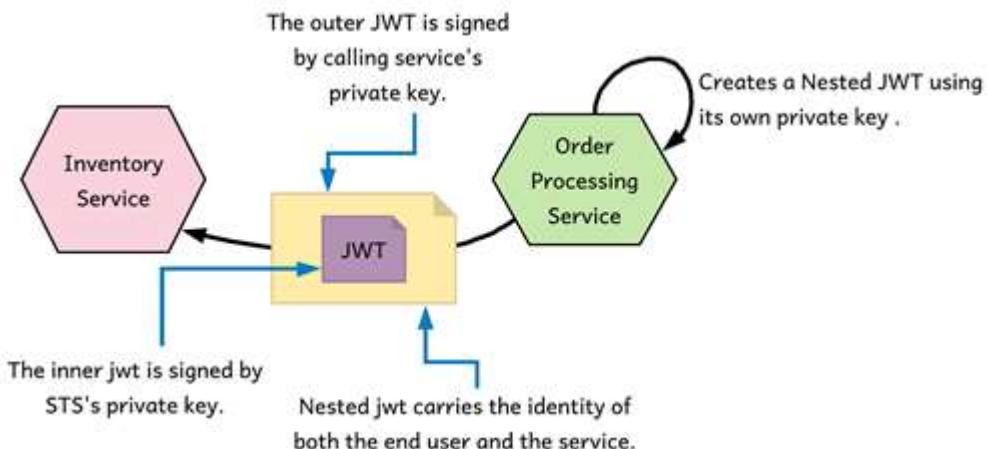


Figure 7.5 Nested JWT. The Order Processing microservice creates its own JWT and embeds in it the JWT it got from the upstream microservice (or the API gateway).

7.2 Setting up an STS to issue a JWT

In this section, you see how to set up an STS to issue a JWT. You're going to use this JWT to access a secured microservice. The source code related to all the examples in this chapter is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, inside the `chapter07` directory. The source code of the STS, which is a Spring Boot application developed with Java, is available in the `chapter07/sample01` directory. This STS is a simple STS and not production-ready. Many open-source and proprietary identity management products can serve as STSes in a production microservices deployment. Run the following Maven command from the `chapter07/sample01` directory to build the STS. If everything goes well, you should see the `BUILD SUCCESS` message at the end:

```
\> mvn clean install  
[INFO] BUILD SUCCESS
```

To start the STS, run the following command from the same directory. The STS by default starts on HTTPS port 8443 and make sure that you have no other services running on the same port. When the server boots up, it prints the time it took to boot up on the terminal:

```
\> mvn spring-boot:run  
INFO 30901 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):  
8443 (https)  
INFO 30901 --- [main] c.m.m.ch07.sample01.TokenService : Started TokenService in  
4.729 seconds (JVM running for 7.082)
```

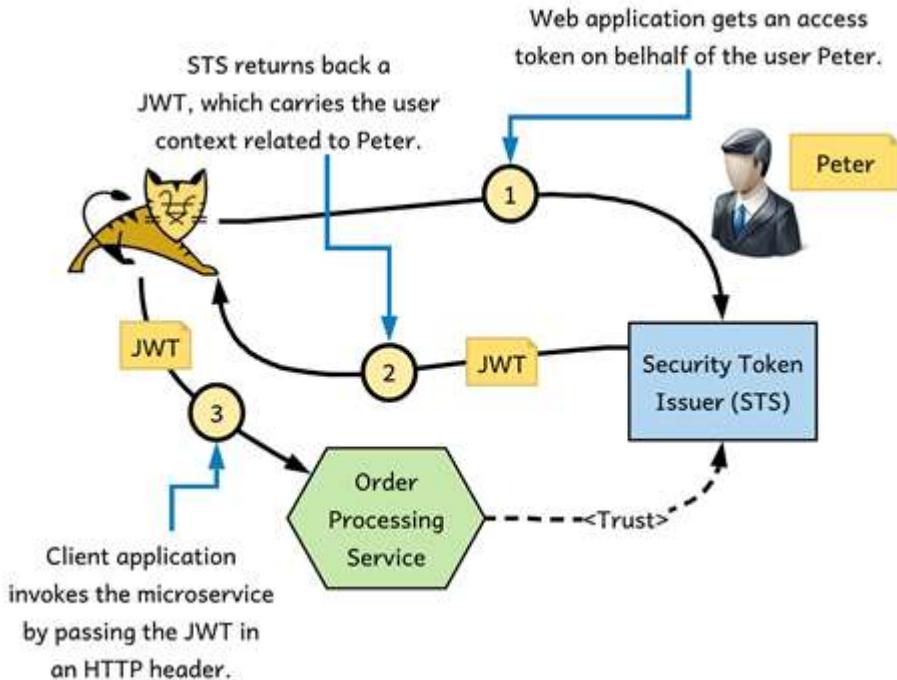


Figure 7.6 STS issues a JWT access token to the web application

Run the following curl command, which talks to the STS and gets a JWT. You should be familiar with the request, which is a standard OAuth 2.0 request following the password grant type. (In chapter 3, we discussed OAuth 2.0 grant types in detail.) You can think of this request as being generated by an external application, such as a web application, on behalf of an end user. In this example the client ID and secret represent the web application, and the username and password represent the end user. Figure 7.6 illustrates this use case. For simplicity, we have removed the API gateway from figure 7.6:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" https://localhost:8443/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJwZXRlcisImF1ZCI6IiouvZNNvbW0uY29tIiwidXNlci19uYW1lIjoicGV0ZXiiLCJzY29wZSI6WyJmb28iXSwiaXNzIjoic3RzMvjb21tLmNvbSIsImV4cCI6MTUzMzI4MDAyNCwiawF0IjoxNTMzMjc5OTY0LCJhdXRob3JpdGllcyI6WyJST0xXF1VTRViXSwanRpIjoiYjjMzkxZjItMWI4MC00ZTgzLTh1YjeTNGE1ZmZmNDR1NjkyIiwiY2xpZW50X21kIjoiMTAxMDEwMTAifQ.MBPq2ngesDB3eCjA0g_ZZdsTd7_Vw4aRocS-ig-
```

```
UHa92xe4LvEl7vADr7SUxPuWrCSre4VkJMwN8uc7KAxJYWH2i0Hfb5haL3jP7074PoCRIKzoSoEB6ZJu7VhW5TVY
4hsOJXWe1dHqPccHTJNKbl0UWBD-yGYnnRyMG47wmQb2MManYURFCFKwFzd03eE0z0BRV5BX-
PsyESgK6qwOV5C6MErVe_Ga_dbVjUR5BGjggjMDlmCoDf403gXK2ifzh_PY1Ggx9eHKPZiq9T1l3yWSvribNgIs9
donciJHh6Nsxt_SFyg7gS-CD66Pg0uA8YRJ5gg3vW6kJVtqsgS8oMYja",
"token_type": "bearer",
"refresh_token": "",
"expires_in": 1533280024,
"scope": "foo"}
```

Following is the payload of the decoded JWT access token:

```
{
    "sub": "peter",
    "aud": "* .ecomm .com",
    "user_name": "peter",
    "scope": [
        "foo"
    ],
    "iss": "sts .ecomm .com",
    "exp": 1533280024,
    "iat": 1533279964,
    "authorities": [
        "ROLE_USER"
    ],
    "jti": "b2c391f2-1b80-4e83-8eb1-4a5fff44e692",
    "client_id": "10101010"
}
```

You can find more details about the STS configuration and the source code in the README file available inside the `chapter07/sample01` directory. In the next section, you see how to use this JWT to access a secured microservice.

7.3 Securing microservices with JWT

In this section, you first secure a microservice with JWT and then use a curl client to invoke it with a JWT obtained from the STS, which you set up in the preceding section. You can find the complete source code related to this example in the `chapter07/sample02` directory. Here, you'll build the Order Processing microservice in Java with Spring Boot and secure it with JWT. First, build the project with the following Maven command from the `chapter07/sample02` directory. If everything goes well, you should see the BUILD SUCCESS message at the end:

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

To start the Order Processing microservice, run the following command from the same directory. When the server boots up, it prints the time that it took to boot up on the terminal. Per the default setting, the Order Processing microservice runs on HTTPS port 9443:

```
\> mvn spring-boot:run
INFO 32024 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
9443 (https)
INFO 32024 --- [main] c.m.m.ch07.sample02.OrderProcessingApp : Started OrderProcessingApp
```

```
in 6.555 seconds (JVM running for 9.62)
```

Now try to invoke the service with the following curl command, with no security token. As expected, the service returns an error message:

```
\> curl -k https://localhost:9443/orders/11
{"error":"unauthorized","error_description":"Full authentication is required to access this
resource"}
```

To invoke the Order Processing microservice with proper security, you need to get a JWT from the STS, using the following curl command. This example assumes that the security token service discussed in the preceding section still runs on HTTPS port 8443. For clarity, we removed the long JWT in the response and replaced it with the value `jwt_access_token`:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
application/x-www-form-urlencoded; charset=UTF-8" -k -d
"grant_type=password&username=peter&password=peter123&scope=foo"
https://localhost:8443/oauth/token

{
"access_token":"jwt_access_token",
"token_type":"bearer",
"refresh_token":"",
"expires_in":1533280024,
"scope":"foo"
}
```

Now try to invoke the Order Processing microservice with the JWT you got from the curl command. Set the same JWT we got, in the HTTP Authorization Bearer header, using the following curl command, and invoke the Order Processing microservice. Because the JWT is little lengthy, you can use a small trick when using the curl command. First, export the JWT to an environmental variable (`TOKEN`). Then use that environmental variable in your request to the Order Processing microservice.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" https://localhost:9443/orders/11
{"customer_id":"101021","order_id":"11","payment_method":{"card_type":"VISA","expiration":"01
/22","name":"John Doe","billing_address":"201, 1st Street, San
Jose, CA"},"items":[{"code":"101","qty":1},{"code":"103","qty":5}],"shipping_address":"2
01, 1st Street, San Jose, CA"}
```

You can find more details about the Order Processing microservice configuration and the source code inside the `chapter07/sample02` directory.

7.4 Using JWT as a data source to do access control

The example in this section is an extension of the example in the section 7.3. Here, you use the same codebase to enforce access control at the Order Processing microservice's end by using the data that comes with the JWT itself. If you're already running Order Processing microservice, first take it down (but keep the STS running). Open the file

sample02/src/main/java/com/manning/mss/ch07/sample02/service/OrderProcessingService.java and uncomment the method-level annotation, @PreAuthorize("#oauth2.hasScope('bar')") from the getOrder() method so that the code looks like the following.

```
@PreAuthorize("#oauth2.hasScope('bar')")
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public ResponseEntity<?> getOrder(@PathVariable("id") String orderId) {
}
```

Rebuild and spin up the Order Processing microservice, using the following two Maven commands from the chapter07/sample02 directory:

```
\> mvn clean install
\> mvn spring-boot:run
```

After the service boots up, you need to get a JWT again from the STS and use it to access the Order Processing microservice. One important thing to notice is that when the client application talks to the STS, it's asking for an access token for the scope `foo`. You'll find the value `foo` in both the curl request and the response. Also, if you decode the JWT in the response from the STS, that too includes an attribute called `scope` with the value `foo`:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
  application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=password&username=peter&password=peter123&scope=foo"
  https://localhost:8443/oauth/token

{
  "access_token": "jwt_access_token",
  "token_type": "bearer",
  "refresh_token": "",
  "expires_in": 1533280024,
  "scope": "foo"
}
```

Now you have a JWT with the `foo` scope from the STS. Try to invoke the Order Processing microservice, which asks for a token with the `bar` scope. Ideally, the request should fail. It fails with an access-denied message, as expected:

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" https://localhost:9443/orders/11
{"error": "access_denied", "error_description": "Access is denied"}
```

Try the same thing with a valid scope. First, request a JWT with the `bar` scope from the STS:

```
: \> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
  application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=password&username=peter&password=peter123&scope=bar"
  https://localhost:8443/oauth/token

{
```

```

"access_token": "jwt_access_token",
"token_type": "bearer",
"refresh_token": "",
"expires_in": 1533280024,
"scope": "bar"
}

```

Now invoke the Order Processing microservice with the right token, and you get a positive response:

```

\> curl -k -H "Authorization: Bearer $TOKEN" https://localhost:9443/orders/11

{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}

```

7.5 Securing service-to-service communication with JWT

Now you have the STS and the Order Processing microservice, which are secured with JWT. In this section, we introduce the Inventory microservice, which is also secured with JWT. We will show you how to pass the same JWT from the client application to the Order Processing microservice and finally (from the Order Processing microservice) to the Inventory microservice. You'll keep both the Order Processing microservice and the STS running, and introduce the new Inventory microservice. First, build the project with the following Maven command from the `chapter07/sample03` directory. If everything goes well, you should see the `BUILD SUCCESS` message at the end:

```

\> mvn clean install
[INFO] BUILD SUCCESS

```

To start the Inventory microservice, run the following command from the same directory. After the server boots up, it prints the time that it took to boot up on the terminal. Per the default setting, the Inventory microservice runs on HTTPS port 10443:

```

\> mvn spring-boot:run
INFO 32024 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
10443 (https)
INFO 32024 --- [main] c.m.m.ch07.sample03.InventoryApp : Started InventoryApp in
6.555 seconds (JVM running for 6.79)

```

Now you need to get a JWT from the STS, using the following curl command, which is the same one you used in the preceding section. For clarity, we removed the long JWT in the response and replaced it with the value `jwt_access_token`:

```

\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=bar" https://localhost:8443/oauth/token

{
"access_token": "jwt_access_token",
"token_type": "bearer",
}

```

```
"refresh_token":"",
"expires_in":1533280024,
"scope":"foo"
}
```

Now try to post an order to the Order Processing microservice with the JWT you got from the preceding curl command. First, export the JWT to an environmental variable (`TOKEN`) and then use that environmental variable in your request to the Order Processing microservice. If everything goes well, the Order Processing microservice validates the JWT, accepts it, and then talks to the Inventory microservice to update the inventory with items. You should be able to find the item numbers printed on the terminal that runs the Inventory microservice.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" -d
  '{"customer_id":"101021","payment_method":{"card_type":"VISA","expiration":"01/22","name":"John Doe","billing_address":"201, 1st Street, San Jose, CA"},"items":[{"code":"101","qty":1},{"code":"103","qty":5}],"shipping_address":"201, 1st Street, San Jose, CA"}' https://localhost:9443/orders
```

You can find more details about the Inventory microservice configuration and the source code inside the `chapter07/sample05` directory.

7.6 Exchanging a JWT for a new one with a new audience

Token exchange is a responsibility of the STS. In this section, you see how to extend the STS you used to issue JWTs to do a token exchange. This STS is simple and not production-ready. Many open-source and proprietary identity management products can serve as STSes (and support exchanging tokens) in a production microservices deployment. You'll use the same codebase you used before for your STS. Run the following Maven command from the `chapter07/sample01` directory to build the STS. If everything goes well, you should see the `BUILD SUCCESS` message at the end. This step is necessary only if you haven't built the project before.

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

To start the STS, run the following command from the same directory. After the server boots up, it prints the time that it took to boot up on the terminal. If you have the STS running already, you can skip this step as well.

```
\> mvn spring-boot:run
INFO 30901 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
  8443 (https)
INFO 30901 --- [main] c.m.m.ch07.sample01.TokenService      : Started TokenService in
  4.729 seconds (JVM running for 7.082)
```

Figure 7.7 illustrates the complete flow of what you're trying to do here. In step-1, the client application gets a JWT access token on behalf of the user.

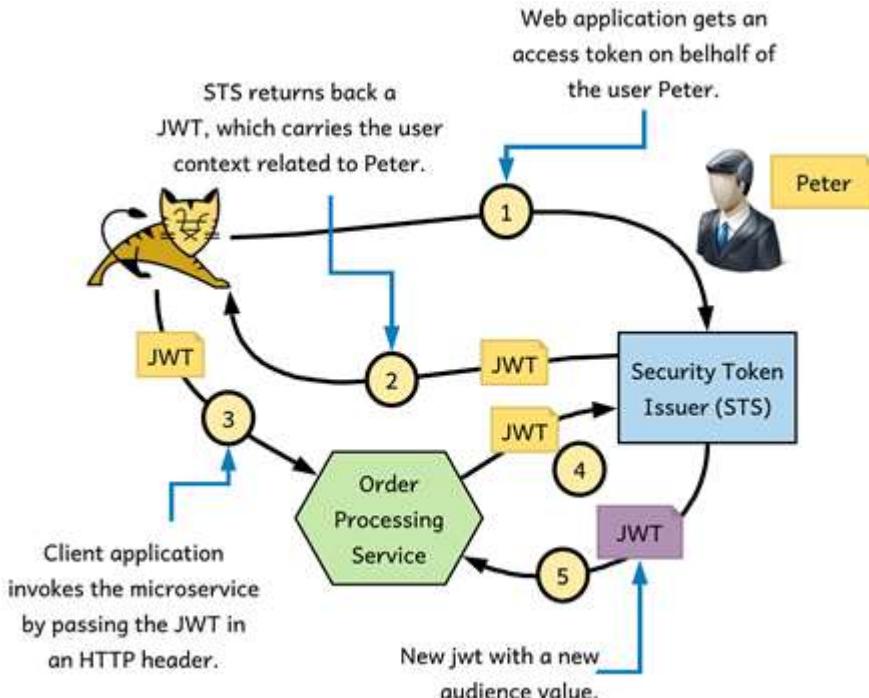


Figure 7.7 Token exchange with STS

Run the following curl command, which talks to the STS and gets a JWT:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=bar" https://localhost:8443/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJwZXRlcIisImF1ZCI6IiouZWNVbW0uY29tIiwidXNlcI9uYW1lIjoicGV0ZXIiLCJzY29wZSI6WyJmb28iXSwiaXNzIjoic3RzMvjb21tLmNvbSISImV4cCI6MTUzMzI4MDAyNCwiaWF0IjoxNTMzMc50TY0LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiYjjMzkxZjItMWI4MC00ZTgzLTlhYjEtNGE1ZmZmNDR1NjkyIiwiY2xpZW50X21kIjoiMTAxMDEwMTAifQ.MBPq2ngesDB3eCjAQg_ZZdsTd7_Vw4aRocS-ig-UHa92xe4LvE17vADr7SUxPuIrcSre4VkmNw8uc7KAxJYWH2i0Hfb5haL3jP7074P0cR1KzoSoEB6ZJu7vhW5TVY4hsOJXWeIdHqPccHTJNKbl0uWBD-yGYnnRyMG47wmQb2MManYURFCFKwFZd03eE0z0BRV5BX-PsyESgK6qwOV5C6MErVe_Ga_dbVjur5B6jggjMD1mCoDf403gXK2ifzh_PY1Ggx9eHKPZiq9T1l3yWSvribNgIs9donciJHh6Nsxt_SFy7gS-CD66PgouA8YRJ5gg3vW6kJVtqsgS8oMYja", "token_type": "bearer", "refresh_token": "", "expires_in": 1533280024,
```

```
"scope": "foo"}
```

Following is the payload of the decoded JWT access token, which carries the audience value *.ecomm.com:

```
{
  "sub": "peter",
  "aud": "*.*.ecomm.com",
  "user_name": "peter",
  "scope": [
    "foo"
  ],
  "iss": "sts.ecomm.com",
  "exp": 1533280024,
  "iat": 1533279964,
  "authorities": [
    "ROLE_USER"
  ],
  "jti": "b2c391f2-1b80-4e83-8eb1-4a5fff44e692",
  "client_id": "10101010"
}
```

Now you're done with steps 1 and 2 (figure 7.7). The client application has a JWT access token. In step 3, it talks to the Order Processing microservice with this JWT. We're going to skip that step and show you what happens in steps 4 and 5—the two most important steps, which show you how the token exchange happens.

Suppose that the Order Processing microservice makes the following call with the JWT it got in step 3. An STS that supports the token-exchange functionality in a standard way must implement the OAuth 2.0 Token Exchange specification (<https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-19>). At this writing, this specification is still in the draft stage (19th draft), but on its way to becoming an RFC. Run the following curl command, which exchanges the JWT, you have with a new one by talking to the STS:

```
\> curl -v -X POST -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=urn:ietf:params:oauth:grant-type:token-
  exchange&subject_token=original_jwt_access_token&subject_token_type=urn:ietf:params:ou
  th:token-type:jwt&audience=inventory.ecomm.com" https://localhost:8443/oauth/token

{
  "access_token": "new_jwt_access_token",
  "issued_token_type": "urn:ietf:params:oauth:token-type:jwt",
  "token_type": "Bearer",
  "expires_in": 60
}
```

In the request to the token endpoint of the STS, you need to send the original JWT under the `subject_token` argument, and the value of the `subject_token_type` argument must be set to `urn:ietf:params:oauth:token-type:jwt`. For clarity, in the preceding curl command, we use the `original_jwt_access_token` identifier to represent the original JWT. Another important argument we use here is the `grant_type`, with the value `urn:ietf:params:oauth:grant-type:token-exchange`.

The STS validates the provided JWT, and if everything looks good, it returns a new JWT with the requested audience value. Now the Order Processing microservice can use this JWT to talk to the Inventory microservice. You can find more details about STS configuration and the source code in the README file available inside the `chapter07/sample01` directory.

NOTE In chapter 12, we discuss how to use JWT to secure service-to-service communications in a service mesh deployment with Istio. Istio is a service mesh developed by Google that runs on Kubernetes. A service mesh is a decentralized application-networking infrastructure between microservices in a particular deployment that provides, resiliency, security, observability, and routing control. We discuss Istio service mesh fundamentals in appendix C.

7.7 Summary

- JSON Web Token (JWT), which provides a way to carry a set of claims or attributes from one party to another in a cryptographically secure way, too plays a key role in securing service-to-service communication in a microservice deployment.
- A JWT can be used to carry the identity of the calling microservice, or the identity of the end user or the system that initiated the request.
- The JWT addresses two main concerns in a microservices security design: securing service-to-service communication and passing end-user context across microservices.
- When the identity of the microservice isn't relevant, but the identity of the end user (system or human) is, you should prefer using JWT to mTLS.
- Having a different/new JWT per each interaction between microservices is a much-secured approach than sharing the same JWT between all the microservices.
- JWT can be used for cross-domain authentication and attribute sharing.
- A self-issued JWT is issued by the microservice itself, and used to authenticate between microservices.
- A *nested JWT* is a JWT that embeds another JWT. It carries the identity of both the calling service and the end user.

8

Securing east/west traffic over gRPC

This chapter covers

- The role of gRPC in inter-service communications in a microservices deployment.
- Securing inter-service communications that happen over gRPC using mutual TLS (mTLS).
- Securing inter-service communications that happen over gRPC using JSON Web Tokens (JWTs).

In chapters 6 and 7 we discussed how to secure communications among microservices with mTLS and JWT. All the examples we used in those chapters assumed the communication between the calling microservice and the recipient microservice happens over HTTP in a RESTful manner with JSON messages. JSON over HTTP is a common way of communicating between microservices. But another school of thought believes using JSON over HTTP to communicate between microservices is not the optimal way. The argument is that human readable, well-structured data interchange format is no value when the communication happens between two systems (or microservices). Which is true since you only need human readable message formats for troubleshooting purposes and not when your systems are running live. Instead of a text-based protocol like JSON, you can use a binary protocol like protocol buffers. It provides a way of encoding structured data in an efficient manner, when communication happens between microservices.

gRPC (<https://grpc.io/>) is an open source remote procedure call framework (or a library), originally developed by Google. It's in fact the next generation of a system called Stubby, which Google has been using internally within Google for over a decade. gRPC achieves efficiency for communication between systems using HTTP/2 as the transport and protocol

buffers as the interface definition language (IDL). In this chapter we discuss how to secure communications between two microservices that happen over gRPC. If you are new to gRPC we recommend you first go through appendix J. It covers gRPC fundamentals.

8.1 Service to service communication over gRPC

In this section of the chapter we discuss the basics of establishing a communication channel between two parties over gRPC. We teach you how to run a simple gRPC client and a server in Java. You can find the source code for this example in the `chapter08/sample01` directory. The use case we use is something that simulates a service-to-service communication. The popular scenario used throughout this book is an example of a retail store. We discuss various use cases of this retail store in different chapters. In this section, we pick a use case where a customer makes an order using our system. When an order is placed, the system needs to update its inventory to make sure the relevant product items that were ordered are removed from the database.

Our system is built in such a way that all major functions are separated into individual microservices. We therefore have microservices for getting product information, processing orders, updating inventory, shipping orders and so on. In this particular use case, you can assume that the Order Processing microservice is exposed to our clients via an API gateway. When a client application makes a call to place an order, the Order Processing microservice takes on the responsibility of making sure the order is properly placed. Within this process it performs a set of coordinated actions, such as processing the payment, updating inventory, initiating the shipping process, and so on. The Inventory microservice is implemented as a gRPC service. Therefore, when the Order Processing microservice needs to update the inventory while processing an order, it needs to make a gRPC request to the Inventory microservice. Figure 8.1 illustrates this use case.

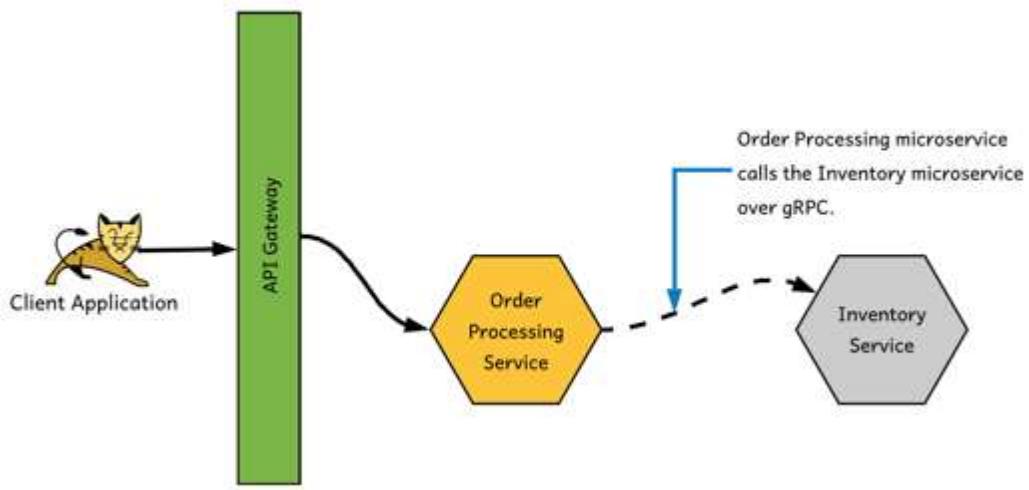


Figure 8.1 – The client application places an order through an API exposed on the API gateway. The client's request is then delegated to the Order Processing microservice. The Order Processing microservice calls the Inventory microservice once the order is placed. The communication between the Order Processing microservice and the Inventory microservice happen over gRPC.

As shown in figure 8.1, the communication between the Order Processing microservice and Inventory microservice happens over gRPC. We will only be focusing on the communication between these two microservices in this section of the chapter.

Let's first look at the interface definition of the Inventory microservice, which uses protocol buffers as its IDL (interface definition language). You can find the IDL of the Inventory microservice at `chapter08/sample01/src/main/proto/inventory.proto` file. Here's the service definition of the Inventory microservice:

```
//The inventory service
service Inventory {
    rpc UpdateInventory (Order) returns (UpdateReply) {}
}
```

This defines the Inventory microservice having a single RPC method named `UpdateInventory`. It accepts a message of type `Order` and returns a message of type `UpdateReply`. The `Order` and `UpdateReply` message types are defined further in listing 8.1.

Listing 8.1 The Protobuf definitions of the Order and UpdateReply messages.

```
//Order type. An order has one or more Line items.
message Order {
    int32 orderId = 1;
    repeated LineItem items = 2;
}

//LineItem. A LineItem consists of a Product with one or more occurrences.
```

```

message LineItem {
    Product product = 1;
    int32 quantity = 2;
}

//Product type.
message Product {
    int32 id = 1;
    string name = 2;
    string category = 3;
    float unitPrice = 4;
}

//The update message sent as a request to the inventory update.
message UpdateReply {
    string message = 1;
}

```

An Order has an id and consists of a collection of LineItem objects. Each LineItem has a quantity and a product. The UpdateReply message has a simple variable of type string. Let's compile our code to auto-generate the service stub and client stub of our InventoryService. To do that, navigate to the chapter08/sample01 directory using your command line client and execute the following command.

```
\> ./gradlew installDist
```

If the stubs are successfully built, you should see a message saying BUILD SUCCESSFUL. You should also see a new directory being created with the name build. This directory has the auto-generated stub classes from the InventoryService proto file.

Open the InventoryGrpc.java file in the build/generated/source/proto/main/grpc/com/manning/mss/ch08/sample01 directory using a text editor or IDE. Within this class, you'll find an inner class named InventoryImplBase with the following descriptor.

```
public static abstract class InventoryImplBase implements io.grpc.BindableService {
```

This is our InventoryService's server stub. To provide the actual implementation of the UpdateInventory RPC, we need to extend this class and override the updateInventory method. Let's take a quick look at how to run our server and client now. To run our server, navigate back to the chapter08/sample01 directory from your command-line client and execute the following command.

```
\> ./build/install/sample01/bin/inventory-server
```

It should output a message saying the server has started on port 50051. Once the server is started, we can now execute our client program. To do that, open up a new terminal window and navigate to the chapter08/sample01 directory and execute the command below.

```
\> ./build/install/sample01/bin/inventory-client
```

If the client runs successfully, you should see a message on your terminal saying INFO: Message: Updated inventory for 1 products. What happened here is that our client program executed the UpdateInventory RPC running on a different port/process (not a different host since we have been using the same machine for both client and server). The server process received the message from the client and executed its UpdateReply method to send back the reply.

Now that we've seen how a typical client server interaction happens, let's take a look at the server and client source code. To understand the client code, open up the chapter08/sample01/src/main/java/com/manning/mss/ch08/sample01/InventoryClient.java file using a text editor or IDE. The client class in this case is initiated using its constructor in the manner shown in listing 8.2.

Listing 8.2 Using the client class's constructor to initiate the client class

```
/**
 * Construct client connecting to InventoryServer
 */
public InventoryClient(String host, int port) {

    this(ManagedChannelBuilder.forAddress(host, port)
        //Channels are secure by default (via SSL/TLS). For the example we
        // disable TLS to avoid needing certificates.
        .usePlaintext()
        .build());
}

/**
 * Construct client for accessing Inventory Server
 */
private InventoryClient(ManagedChannel channel) {
    this.channel = channel;
    inventoryBlockingStub = InventoryGrpc.newBlockingStub(channel);
}
```

What happens here is that when the client is instantiated, a Channel is created between the client and server. A gRPC Channel provides a connection to a gRPC server on a specified host and port. In this particular example we are disabling TLS (transport level security) to keep things simple. You may notice that we explicitly set `usePlaintext()` to indicate that we are not doing TLS or mTLS on this Channel. We look at TLS specifics later on in the chapter. The created Channel is then used to instantiate a stub, named `InventoryBlockingStub`. This stub will then be used to communicate with the server when RPCs are being executed. The reason this is called a **blocking** stub is because in this particular case we are using a stub that blocks the running thread until the server receives a response or raises an error. The other alternative to use is the `InventoryFutureStub`, which is a stub that does not block the running thread and expects the server to respond later in time.

The communication between the client and server happens in the `updateInventory` method in the client class. This method receives an object of type `OrderEntity`, which

contains the details of the confirmed order. It converts the `OrderEntity` object in to its respective RPC object and passes the RPC object to the server.

```
/**
 * Update inventory upon confirming an order
 */
public void updateInventory(OrderEntity order) {

    . . . .
    . . . .

    UpdateReply updateResponse;
    try {
        updateResponse = inventoryBlockingStub.updateInventory(orderBuilder.build());
    } catch (StatusRuntimeException e) {
        logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
        return;
    }
}
```

The `inventoryBlockingStub.updateInventory(..)` statement transports the `Order` object to the server and gets the response from the server. The server code of the inventory server is relatively simpler. To see what the server code looks like, open the `sample01/src/main/java/com/manning/mss/ch08/sample01/InventoryServer.java` file using a text editor or IDE. The start method of this class contains the code that starts the inventory server.

```
private void start() throws IOException {
    /* The port on which the server should run */
    int port = 50051;
    Server server = ServerBuilder.forPort(port)
        .addService(new InventoryImpl())
        .build()
        .start();
    logger.info("Server started, listening on " + port);
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            // Use stderr here since the logger may have been reset by its JVM shutdown hook.
            System.err.println("*** shutting down gRPC server since JVM is shutting down");
            InventoryServer.this.stop();
            System.err.println("*** server shut down");
        }
    });
}
```

This method starts the server on port 50051. It then adds gRPC services to be hosted on this server process. In this particular example, we only add the Inventory service using the class `InventoryImpl`. The `InventoryImpl` is an inner class declared within the same class. It extends the auto generated `InventoryImplBase` class and overrides the `updateInventory` method.

```
static class InventoryImpl extends InventoryGrpc.InventoryImplBase {

    /**
     *
     */
    public void updateInventory(OrderEntity order) {
        . . . .
        . . . .
    }
}
```

```

    * Method that updates the inventory upon receiving a message.
    * @param req - The request
    * @param responseObserver - A handle to the response
    */
@Override
public void updateInventory(Order req, StreamObserver<UpdateReply> responseObserver) {
    UpdateReply updateReply = UpdateReply.newBuilder().setMessage("Updated inventory
for " + req.getItemsCount()
        + " products").build();
    responseObserver.onNext(updateReply);
    responseObserver.onCompleted();
}
}

```

When our client class executed its `updateInventory` method, the client stub transported over the network through the `Channel` that was created between the client and server and executes the `updateInventory` method on the server. In this example, the `updateInventory` method on the server simply replies to the client saying the inventory was updated with a number of items received on the order request. In a typical scenario, this would probably update a database and remove the items that were ordered from the stocks.

This concludes our first example of a simple client to server communication over gRPC. In the next section of the chapter we are going to look other modes of communication that happen between two parties communicating over gRPC and their benefits.

8.2 Securing gRPC service-to-service communication with mTLS

In this section, we look at securing a channel between two parties who communicate over gRPC, using mutual TLS (mTLS). In section 8.1, we discussed a simple communication channel that happens between a client and server over gRPC. We discussed a scenario of a retail store use case where a microservice that was processing orders needed to communicate with the microservice that was supposed to update the inventory. In a traditional monolithic application architecture pattern, processing orders and updating the inventory would have been done via two functions within the same process/service. The scope of the `updateInventory` function would be designed in such a way where it is only physically accessible from the `orders` function only. Figure 8.2 illustrates the monolithic process.

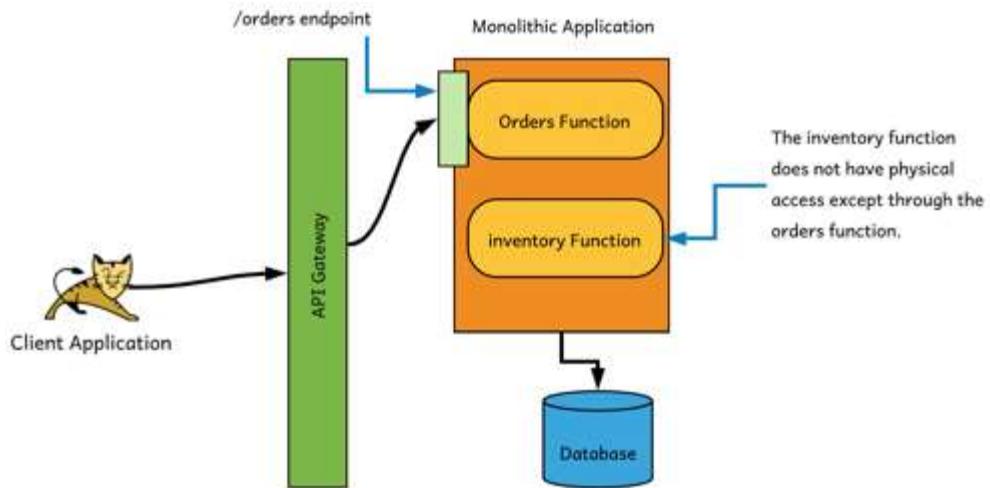


Figure 8.2 – In a monolithic application, functions that are not exposed over a network do not have physical access unless within the application itself.

As you can see from figure 8.2, the only entry point into the monolith web service is through the `/orders` endpoint, which is exposed via the API gateway. The `updateInventory` function is physically inaccessible by anyone else.

However, in microservices architecture, the situation is quite different. The inventory microservice is deployed independently. Therefore, anyone with physical access to the microservice at a network level can invoke its functions. From a use case point of view, we need to prevent this. We need to ensure that our inventory is only updated upon processing an order. We therefore need to ensure that only the Order Processing microservice can execute the functions on the Inventory microservice, even if others have physical access to it. Figure 8.3 illustrates this scenario.

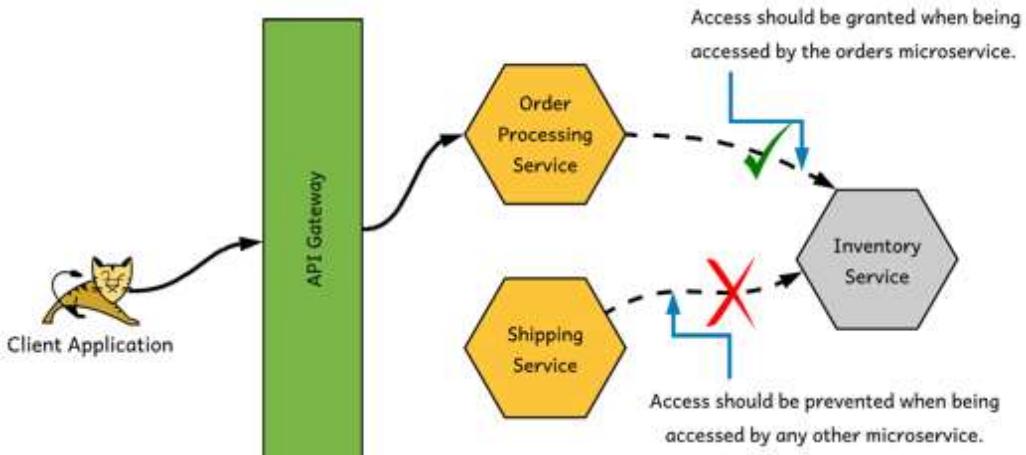


Figure 8.3 – It should only be the Order Processing microservice that can access the inventory microservice. All other accesses should be prevented.

As we discussed in detail in chapter 6, there is where mutual TLS (mTLS) comes into play. mTLS allows us to build an explicit trust between the Order Processing microservice and Inventory microservice using certificates. Whenever a communication happens between the two parties over mTLS, the Order Processing microservice validates that it is actually talking to the Inventory microservice using regular TLS. And the Inventory microservice validates that it is indeed the Order Processing microservice that calls it by validating the certificate of the client (Order Processing microservice).

8.2.1 Running the example with mTLS

In this section, run the same example we ran in section 8.1, but with mutual TLS enabled between the Order Processing microservice and Inventory microservice. You can find the samples for this section in `chapter08/sample02` directory. In addition to the prerequisites defined in section 2.1.1 of chapter 2, you also need to install OpenSSL on your computer. You can check if you already have OpenSSL installed by executing the following command on your command line tool. If you do not want to take the pain of installing OpenSSL, you can also run it as a Docker container, in the way we discussed in section 6.2.1 of chapter 6.

```
\> openssl version
```

The output of the command would indicate the version of OpenSSL on your computer if you have it installed. Check out the samples from the git repo to your computer and use your command line client to navigate to the `sample02` directory. Execute the following command to compile the source code and build the binaries for the client and server.

```
\> ./gradlew installDist
```

If the source is successfully built, you should see a message saying `BUILD SUCCESSFUL`. You should also see a directory named `build` being created inside the `sample02` directory.

As the next step we need to create the certificates and keys required for our client and server. In chapter 6 we discussed in detail the fundamentals of mutual TLS, including the steps required to create keys and certificates. We therefore will not repeat the same steps here. The `mkcerts.sh` script will create the required certificates for us in one go. Note that you need to have OpenSSL installed on your computer for the script to work. Execute the script by using the command below.

```
\> ./mkcerts.sh
```

The above command creates the required certs in the `/tmp/sslcert` directory. Once the certs are successfully created, we can now start our server, which hosts the Inventory microservice using the command below.

```
\> ./build/install/sample02/bin/inventory-server localhost 50440 /tmp/sslcert/server.crt  
/tmp/sslcert/server.pem /tmp/sslcert/ca.crt
```

If the server starts successfully, you should see the following message.

```
INFO: Server started, listening on 50440
```

As you can observe from the command we just executed, we pass in 5 parameters to the process. Their values, along with their usages, are listed below.

- `localhost` – The Host address to which the server process binds to
- `50440` – The port on which the server starts
- `/tmp/sslcert/server.crt` – The certificate chain file of the server.
- `/tmp/sslcert/server.pem` – The private key file of the server.
- `/tmp/sslcert/ca.crt` – The trust store collection file, which contains the certificates to be trusted by the server.

NOTE If you need to know the importance of the certificate files and private key file listed above, please go through chapter 6 where their importance is explained in detail.

To start the client process, open a new terminal window and navigate to the `chapter08/sample02` directory and execute the command below.

```
\> ./build/install/sample02/bin/inventory-client localhost 50440 /tmp/sslcert/ca.crt  
/tmp/sslcert/client.crt /tmp/sslcert/client.pem
```

Similar to how we executed the server process, we need to pass in similar parameters to the client process as well. Following lists down what they mean.

- `localhost` – The host address of the server.
- `50440` – The port of the server.

- /tmp/sslcert/ca.crt – The trust store collection file, which contains the certificates to be trusted by the client.
- /tmp/sslcert/client.crt – The certificate chain file of the client.
- /tmp/sslcert/client.pem – The private key file of the client.

If the client executes successfully, you should see a message on your terminal saying,

```
INFO: Message: Updated inventory for 1 products
```

Let's look at the source code to understand how we enabled mutual TLS on our server and client processes. To look at the server code, open the sample02/src/main/java/com/manning/mss/ch08/sample02/InventoryServer.java file using a text editor or IDE. Let's first take a look at the start method, which starts the server process.

Listing 8.3 Code listing of the server start process

```
private void start() throws IOException {
    server = NettyServerBuilder.forAddress(new InetSocketAddress(host, port))
        .addService(new InventoryImpl())
        .sslContext(getSslContextBuilder().build())
        .build()
        .start();
    logger.info("Server started, listening on " + port);
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            // Use stderr here since the logger may have been reset by its JVM shutdown hook.
            System.err.println("*** shutting down gRPC server since JVM is shutting down");
            InventoryServer.this.stop();
            System.err.println("*** server shut down");
        }
    });
}
```

As you can see from listing 8.3, we start the server process by binding it to the host, which is the first argument we passed to the server starting command. The process is started on the passed-in port. We add the `InventoryImpl()` service to the server process using the `addService()` function. The important line to look at here is the one in bold text, which sets the `sslContext` to our server. This `sslContext` that is passed in contains information such as the server certificate file, private key file and trust store file. Let's also look at the `getSslContextBuilder()` method.

```
private SslContextBuilder getSslContextBuilder() {
    SslContextBuilder sslClientContextBuilder = SslContextBuilder.forServer(new
        File(certChainFilePath),
        new File(privateKeyFilePath));
    if (trustCertCollectionFilePath != null) {

        sslClientContextBuilder.trustManager(new File(trustCertCollectionFilePath));

        sslClientContextBuilder.clientAuth(ClientAuth.REQUIRE);
    }
}
```

```

    }
    return GrpcSslContexts.configure(sslClientContextBuilder,
        SslProvider.OPENSSL);
}
}

```

This method sets some context variables, which define the behavior of the server process. By setting `sslClientContextBuilder.clientAuth(ClientAuth.REQUIRE)`, we are mandating that the client application presents its certificate to the server for verification (mandating mTLS). Let's now look at the client code and see what changes we had to do to make it work over mTLS. First let's recall a part of the client code from section 8.2 under code listing 8.3. Here's how we implemented the constructor of the client class in section 8.2.

```

public InventoryClient(String host, int port) {
    this(ManagedChannelBuilder.forAddress(host, port)
        // Channels are secure by default (via SSL/TLS). For the example we disable TLS
        to avoid
        // needing certificates.
        .usePlaintext()
        .build());
}

```

You may notice that we explicitly set `usePlaintext()` to indicate that we are not doing TLS or mTLS on this channel. The same constructor has been enhanced as follows to enable TLS/mTLS.

```

public InventoryClient(String host, int port, SslContext sslContext) throws SSLEException {
    this(NettyChannelBuilder.forAddress(host, port)
        .negotiationType(NegotiationType.TLS)
        .sslContext(sslContext)
        .build());
}

```

We now have a `sslContext` being set instead of saying `usePlaintext()`. The `sslContext`, as with how the `sslContext` of the server was set, bears information of the trust store file, client certificate chain file and the private key. The listing 8.4 shows how the client `sslContext` is built.

Listing 8.4 How the client `sslContext` is built

```

private static SslContext buildSslContext(String trustCertCollectionFilePath, String
    clientCertChainFilePath, String clientPrivateKeyFilePath) throws SSLEException {

    SslContextBuilder builder = GrpcSslContexts.forClient();
    if (trustCertCollectionFilePath != null) {
        builder.trustManager(new File(trustCertCollectionFilePath));
    }
    if (clientCertChainFilePath != null && clientPrivateKeyFilePath != null) {
        builder.keyManager(new File(clientCertChainFilePath), new
            File(clientPrivateKeyFilePath));
    }
    return builder.build();
}

```

The `if` conditions in listing 8.4 are to check if the relevant arguments have been provided at the point of starting the client process. The client is built in a way where you could run it with mTLS, with TLS and without anything at all. Let us try to run the client with just TLS (no mTLS) and see how our server responds. Assuming your server process is still running, re-run the client process as below (fewer arguments).

```
\> ./build/install/sample02/bin/inventory-client localhost 50440 /tmp/sslcert/ca.crt
```

Here, we are only providing the trust store collection to the client, which makes the client work in TLS mode only. After executing this command, you should notice an error saying, connection refused. This is because on the server side we have mandated mTLS as we just discussed. In this mode, the server expects the client to present its certificate information and we have executed the client process without specifying its own certificate and primary key information.

8.3 Securing gRPC service-to service communications with JWT

In this section, we discuss in detail how we can secure a communications channel between two parties that is happening over gRPC using JWTs. In chapter 7, we discussed in detail securing service-to-service communications using JWTs. The same fundamentals apply throughout this section as well. We'll use the knowledge we gained in chapter 7 to understand how the same concepts apply in the context of a gRPC communications channel. Further in chapter 7, under the section 7.1 we discuss the use cases for securing microservices with JWT, the benefits of JWT over mTLS as well as how both JWT and mTLS complement each other. In this section we use a practical scenario to demonstrate how you can use JWTs effectively over gRPC.

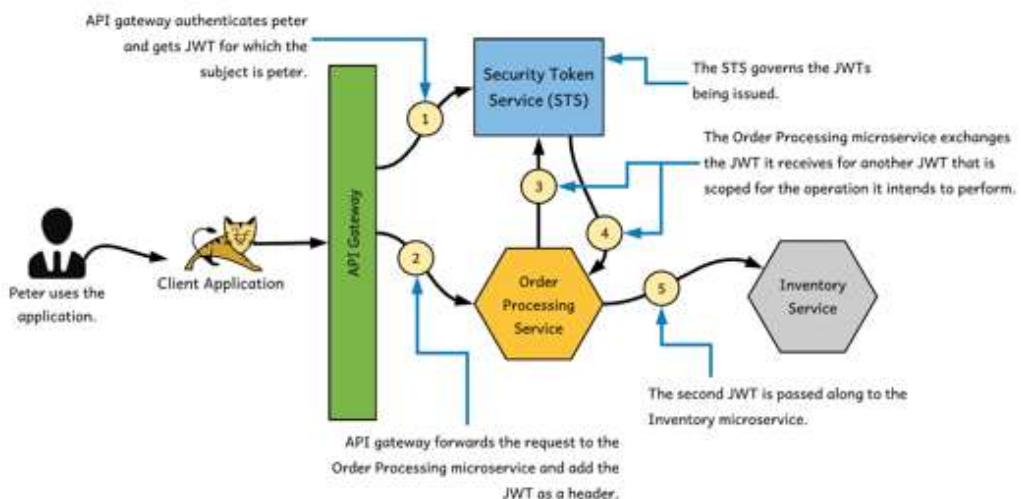


Figure 8.4 – The JWT received by the Order Processing microservice is exchanged for a secondary JWT, which is scoped to access the relevant operations on the inventory microservice.

As shown in figure 8.4 we have the Order Processing microservice, which exchanges the JWT it receives from the client application for another (second) JWT with the help of an STS. This new JWT will then be passed along to the Inventory microservice. When the Order Processing microservice does the JWT exchange in step-3, the STS has complete control over the JWT being issued. The STS can therefore decide who the audience of the JWT should be, what other information to include/exclude in the JWT, the scopes of the JWT and so on. By specifying the scopes associated to the JWT the STS determines which operations this JWT is allowed to perform on the respective audience.

RUNNING THE SAMPLE

You can find the source code related to the sample we discuss in this section inside the chapter08/sample03 directory. In the sample03 directory you'll find two directories: one named `sts` and the other named `client_server`. Let's start by compiling and running the STS for us to get a JWT.

Open a command line client tool and navigate to the `sample03/sts` directory and execute the command below to compile the source code of the STS.

```
\> mvn clean install
```

If the build is successful, you'll see a message saying `BUILD SUCCESS`. Run the following command to spin up the STS.

```
\> mvn spring-boot:run
```

If the STS starts properly, you should see a message similar to `Started TokenService in 3.875 seconds`. After the STS got started, open a new terminal tab and use the following command to generate a JWT.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" https://localhost:8443/oauth/token
```

The above command makes a token request to the STS using Peter's credentials. And you should get an access token in JWT form as the response, as shown below. Note that we've omitted the long string that represents the JWT (which is the value of the `access_token` parameter) and replaced it with `jwt_token_value` for brevity. You can inspect the JWT token via <https://jwt.io>. It will show you the decoded JWT string.

```
{"access_token":"jwt_token_value","expires_in":5999,"scope":"foo","jti":"badc4a65-b6d6-4a1c-affc-3d0565cd2b55"}
```

Once we have a JWT access token, we can use it to access the Inventory microservice. To begin our gRPC service, first navigate to the `chapter08/sample04/client_server` directory and execute the following command.

```
\> ./gradlew installDist
```

If the program compiles successfully, you should see a message saying BUILD SUCCESSFUL. Let's start the server process by executing the following command.

```
\> ./build/install/client_server/bin/inventory-server localhost 50440 /tmp/sslcert/server.crt  
/tmp/sslcert/server.pem
```

If the server starts successfully, you should see a message saying INFO: Server started, listening on 50440. Note that we are using TLS only (not mTLS) for this sample. We are using JWT as the client verification process. The next step would be to start the client process. Before doing that, open a new terminal window, navigate to the chapter08/sample03/client_server directory, and set the value of the access token as an environment variable using the following command. Please make sure to replace jwt_token_value with your token's actual value.

```
\> export TOKEN=jwt_token_value
```

Next, execute the following command on the same terminal window to execute the gRPC client. Note that we are providing the value of the obtained JWT access token to the gRPC client as an argument.

```
\> ./build/install/client_server/bin/inventory-client localhost 50440 /tmp/sslcert/ca.crt  
$TOKEN
```

If the client executes successfully, you should see a message saying INFO: Message: Updated inventory for 1 products. Let us now execute the same gRPC client without providing the JWT access token as an argument. Execute the following command on the same terminal window.

```
\> ./build/install/client_server/bin/inventory-client localhost 50440 /tmp/sslcert/ca.crt
```

You should see an error message saying WARNING: RPC failed: Status {code=UNAUTHENTICATED, description=JWT Token is missing from Metadata, cause=null}. This is because the server is written to expect a valid client JWT, and when it doesn't receive it, it throws an error.

UNDERSTANDING THE CODE

Unlike in HTTP, gRPC doesn't have headers. gRPC supports sending metadata between client and server. Metadata is a key-value pair map where the key is a string and the value can be a string or in binary form. Keys that contain binary data as the values need to be suffixed with the string -bin. For example, key-bin. In the example we just executed, we used metadata to transfer the JWT access token from client to server. We used something called a ClientInterceptor, which intercepts the messages being passed from client to server on the channel to inject the JWT access token as gRPC metadata. The code for the ClientInterceptor can be found in the chapter08/sample03/client_server/src/main/java/com/manning/mss/ch08/sam

ple03/JWTClientInterceptor.java file. Its `interceptCall` method is the one that's executed on each message passed. Listing 8.5 shows what it looks like.

Listing 8.5 Method that executes on each message to validate the JWT

```
@Override
public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(
    MethodDescriptor<ReqT, RespT> methodDescriptor, CallOptions callOptions,
    Channel channel) {

    return new ForwardingClientCall.SimpleForwardingClientCall<ReqT, RespT>(
        channel.newCall(methodDescriptor, callOptions)) {
        @Override
        public void start(Listener<RespT> responseListener, Metadata headers) {
            headers.put(Constants.JWT_KEY, tokenValue);
            super.start(responseListener, headers);
        }
    };
}
```

Notice the text in bold, which is where we set the value of the JWT access token to request metadata. To understand how the interceptor class is set to the gRPC client, take a look at the following method, which is found in the `InventoryClient` class under the same package as the `clientInterceptor` class.

```
public InventoryClient(String host,
                      int port,
                      SslContext sslContext,
                      JWTClientInterceptor clientInterceptor) throws SSLEException {

    this(NettyChannelBuilder.forAddress(host, port)
        .negotiationType(NegotiationType.TLS)
        .sslContext(sslContext)
        .intercept(clientInterceptor)
        .build());
}
```

As shown in the line in bold text, the client interceptor is set to the channel being created between the client and server. We follow a similar approach on the server side to validate the JWT access token. The server too uses an interceptor, called `ServerInterceptor` to intercept incoming messages and perform validations on them. The `serverInterceptor` class can be found in the `chapter08/sample03/client_server/src/main/java/com/manning/mss/ch08/sample03/JWTServerInterceptor.java`. Its `interceptCall` method, shown below is executed on each message. We retrieve the JWT access token from the metadata and validate it at this point.

```
@Override
public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(ServerCall<ReqT, RespT>
    serverCall,
    Metadata metadata,
    ServerCallHandler<ReqT, RespT> serverCallHandler) {
```

```

// Get token from Metadata
String token = metadata.get(Constants.JWT_KEY);
System.out.println("Received Token: " + token);

if (!validateJWT(token)) {
    serverCall.close(Status.UNAUTHENTICATED.withDescription("JWT Token is missing from
    Metadata"), metadata);
    return NOOP_LISTENER;
}

return serverCallHandler.startCall(serverCall, metadata);
}

```

The text in bold shows how we obtain the JWT access token from the request metadata and then validate it. This way we can use gRPC metadata to pass in an access token from client to server.

8.4 Summary

- In a microservices deployment, given that there are many interactions that happen over the network between microservices, JSON over HTTP/1.1 is not efficient enough.
- gRPC operates over HTTP/2, which is significantly more efficient than HTTP/1.1 due to request response multiplexing, binary encoding, and header compression.
- Unlike in HTTP/1.1, HTTP/2 supports bidirectional streaming, which is beneficial in microservice architectures.
- gRPC supports mTLS, which you can use to secure communication channels between microservices.
- mTLS does not necessarily address the full spectrum of security we need to ensure on microservice architectures; we therefore need to resort to JWTs in certain cases.
- Unlike HTTP, since gRPC does not have a concept of headers, we have to use metadata fields in gRPC to send JWTs.
- The client interceptors and server interceptors available in gRPC help to send JWTs from clients to servers and to validate them.

9

Securing reactive microservices

This chapter covers

- The need for reactive microservices in a microservices deployment and its benefits
- Using Apache Kafka as a message broker for inter-service communications
- Using Transport Layer Security (TLS) in Kafka to secure messages in transit
- Using mutual TLS (mTLS) in Kafka to authenticate microservices connecting to Kafka
- Controlling access to Kafka topics using access control lists

In chapter 6 and chapter 7 we discussed how to secure service-to-service communications with mutual Transport Layer Security (mTLS) and JSON Web Token (JWT). Chapter 8 extended that discussion and explained how mTLS and JWT can be used to secure communications happening over gRPC. In all those cases, the examples we used, assumed synchronous communication between the calling microservice and the recipient microservice. The security model that you develop to protect service-to-service communications should consider how the actual communication takes place between microservices: synchronously or asynchronously.

In most cases, synchronous communication happens over HTTP. Asynchronous communication can happen over any kind of messaging system, such as RabbitMQ, Kafka, ActiveMQ, or even Amazon SQS. In this chapter we discuss how to use Apache Kafka as a message broker, which enables microservices to communicate with each other in an event-driven fashion and how to secure the communication channels. Kafka is the most popular messaging system, which is used in many microservice deployments. If you are interested in learning more about Kafka, we would recommend you have a look at the book, *Kafka in Action* (Manning Publications, 2020) by Dylan Scott.

9.1 Why reactive microservices?

In this section we discuss the need to have reactive microservices in your microservices deployment. A microservice is considered to be reactive when it can react to events that occur in a system without explicitly being called by the event originator. Or in other words, the recipient microservice is decoupled from the calling microservice. The microservice, which generates events, does not necessarily need to know which microservices do consume those events. Let us take the example of a typical order placement scenario. As we discussed in chapter 8, multiple actions take place when an order is placed, as listed below.

- Preparing the invoice and processing the payment.
- Updating the inventory to reduce the items in stock.
- Processing the shipment to the customer who placed the order.
- Sending an email notification to the customer regarding the status of the order.

In a typical microservices deployment, each of these actions is performed by an independent microservice. This way each operation becomes independent from each other and a failure in one doesn't cause an impact to others. For example, if a bug in the shipping microservice causes it to go out of memory, it doesn't impact the rest of the functionality with respect to processing the order and the order can still be completed. In previous chapters we looked at how the orders microservice becomes the triggering point for the rest of the actions that takes place. When an order is processed by the Order Processing microservice it initiates the rest of the actions that take place. Such as the inventory update, initializing the shipment and so on. This way, the Order Processing microservice became the orchestrator for the rest of the actions related to processing an order. Figure 9.1 illustrates this pattern further.

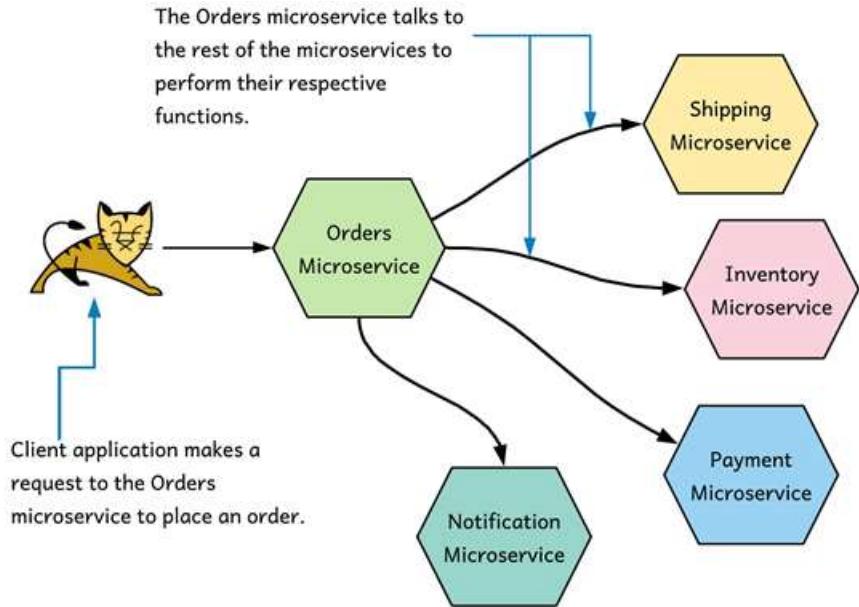


Figure 9.1 – The orders microservice talks to other microservices to perform functions related to processing an order. These are related to processing the payment of the order, updating the inventory and so on.

As you can see in figure 9.1, the Order Processing microservice calls out to all the other microservices that perform their respective actions. There are two main types of actions in these types of interactions that happen between microservices: the synchronous actions and the asynchronous actions.

The synchronous actions are the types that need to be completed in real-time for the order to be considered successful (completed). Such as the payment processing. The payment needs to be successfully completed before the order can be considered successfully completed. And also the Order Processing microservice needs to update the system's database to record an order successfully.

The asynchronous actions are the types can be performed later. The order can be considered complete even if these actions are not performed in real time. Such as the update of the inventory, processing of the shipment and sending an email to the customer for example. These are the actions that we can perform offline, asynchronous to the main operation.

For the Order Processing microservice to be the orchestrator, which invokes the respective microservices that perform the rest of the actions related to processing an order, it needs to know how to invoke these microservices, such as the connectivity information of the relevant microservices, the parameters that need to be passed in, and so on. Take a look at listing 9.1

for a code sample of how the Order Processing microservice would have to invoke functions to trigger the rest of the actions to be performed.

Listing 9.1 – Orders microservice calls other microservices synchronously

```
try {
    updateResponse = inventoryStub.updateInventory(orderBuilder.build());
} catch (StatusRuntimeException e) {
    logger.log(Level.WARNING, "Unable to update inventory. RPC failed: {0}", e.getStatus());
    return;
}

try {
    updateResponse = shippingStub.makeShipment(orderBuilder.build());
} catch (StatusRuntimeException e) {
    logger.log(Level.WARNING, "Unable to make shipment. RPC failed: {0}", e.getStatus());
    return;
}

try {
    updateResponse = notificationsStub.sendEmail(orderBuilder.build());
} catch (StatusRuntimeException e) {
    logger.log(Level.WARNING, "Unable to send email to customer. RPC failed: {0}",
        e.getStatus());
    return;
}
```

Imagine a situation where we would need a new action to be performed when an order is being processed. Imagine that we introduce a new feature which tracks the buying patterns of each customer so that we can provide recommendations to other buyers. If we introduce this new feature as a separate microservice, we would now need to update the Order Processing microservice to invoke this new microservice as well. This causes us to make code changes to the Order Processing microservice and redeploy it in order for the new feature to come into effect.

As you may have observed, the Order Processing microservice has eventually become a center of excellence where it becomes a trigger for a number of actions being performed. It is responsible to trigger the shipment, send emails and update the inventory. While all of these actions must be performed for an order to be successfully completed, the fact that the Order Processing microservice performs many actions is not ideal as per the principles of microservice architecture. The two main things that are mandatory for the order to be successfully recorded are the payment processing and the database update, which records the transaction. The rest of the actions can be performed asynchronously once the two main actions are successfully completed. This is where reactive microservices become useful.

Reactive microservices work in such a way that the microservices are attentive to events that occur in the system and act accordingly based on the type of event that occurs. In this particular example, the Order Processing microservice, upon completion of recording the order in the system, would emit an event to the system with the details of the order. The rest of the microservices, which are paying attention to these events would receive the order event and react to it accordingly. By reacting, what they do is to perform their respective operations upon

receiving the event. For example, when the shipping microservice receives the order event it would perform the actions necessary to ship the order to the customer who placed it.

Reactive microservices create a loose coupling between the source microservice that initiates the event and the target microservices that receive and react to the event. As we saw in figure 9.1, in the older/traditional way of performing these actions there is a direct link from the Order Processing microservice to the rest of the microservices such as Inventory, Shipping and so on. With reactive microservices, this link becomes indirect. This happens by introducing a message broker solution into our microservices deployment. See figure 9.2 for an illustration of this solution.

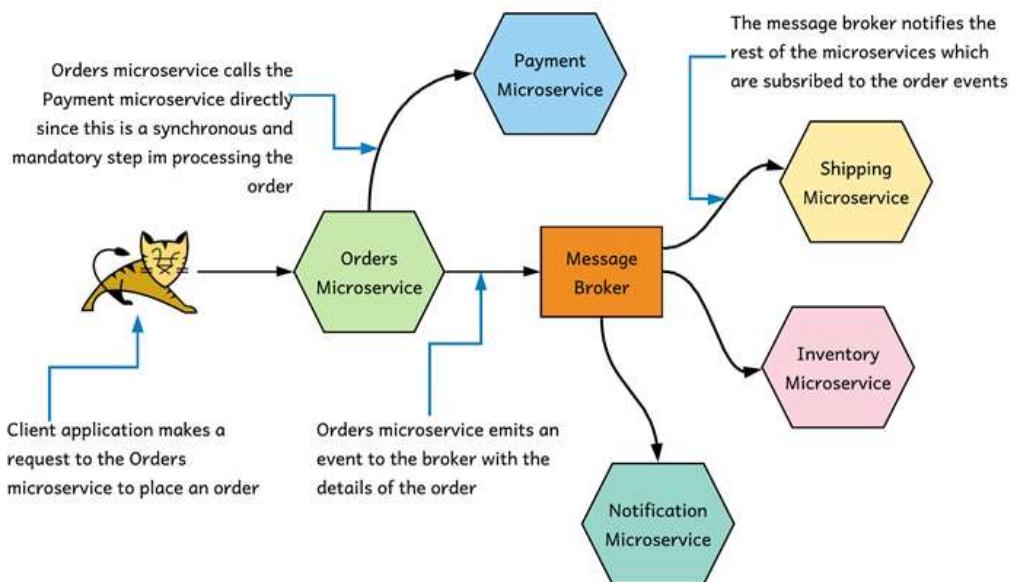


Figure 9.2 – Introducing a message broker into the architecture. The Order Processing microservice calls the Payment microservice directly since payment is a mandatory and synchronous step in processing the order. It then emits an event to the message broker which deliver the order details to the rest of the microservices asynchronously. This makes the link between the Order Processing microservice and the rest indirect.

As you see in figure 9.2 the Order Processing microservice emits an event (sends a message) to the message broker. All other microservices interested in knowing about the orders being processed in the system are subscribed to the particular topic on the broker.

In messaging systems, the message producers publish messages to either a queue or topic. Consumers of the message subscribe to the queues or topics of interest to receive messages. A topic is used when we want all subscribers

of the topic to receive the event and process. A queue is used when an event needs to be processed by one subscriber only, the first subscriber to receive it.

Upon receiving an order event, the consumer microservices start executing their processes to complete the tasks they are responsible for. For example, when the Inventory microservice receives the order event, it starts executing its code to update the inventory as per the details in the order. The key benefit in this architecture is the loose coupling between the source and the target microservices. This loose coupling allows the Order Processing microservice to focus on its main responsibility, which is to ensure the payment is properly processed via the Payment microservice and the recording of the order itself in the system.

Another major benefit in event driven architecture is that it allows us to add new functionality into the system without affecting the current code. Referring to the same example we discussed earlier, imagine we wanted to introduce a feature that tracks a user's buying patterns. As per the older architecture we had to introduce a new microservice which tracks the buying patterns and also change the Order Processing microservice so that it could talk to the Buying History microservice. But with the reactive architecture all we need to do is to make the Buying History microservice aware of the order event by linking it with the message broker. This way the Buying History microservice gets to know the details of each order whenever an order is processed. Hence giving us the ability to add new functionality to the system without having to change and redeploy old code. Figure 9.3 illustrates how the Buying History microservice has been introduced into this architecture.

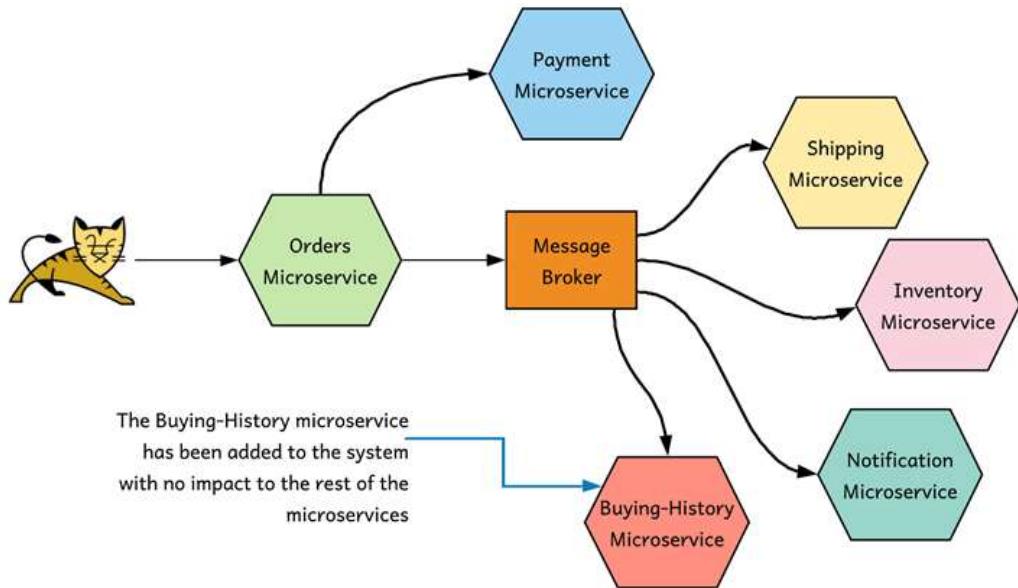


Figure 9.3 – Introducing the buying-history microservice to the system. We were able to add the buying-history microservice to the system and benefit from the capabilities without having to make any changes on the orders microservice or anything else.

9.2 Setting up Kafka as a message broker

In this section we look at how we can set up Apache Kafka as a message broker in our microservices deployment. We discuss how we install Kafka, create a topic on Kafka and transfer a message from a message producer to a receiver. Apache Kafka is an open source distributed streaming platform. It is capable of publishing and subscribing to streams of records (data), similar to that of message queues. It can store streams of data as they receive and process streams of data as they arrive as well. We use Kafka in this section to receive streams of data events from some microservices and deliver them to other microservices in an asynchronous fashion.

To setup Kafka in your local machine, first download it from <https://kafka.apache.org/downloads>. The version we use in this chapter is 2.3.1 (with Scala version 2.1.2), the latest version as of this writing. Once you have downloaded it, extract the zip archive to a location of choice. Use your command line tool to navigate to the location where you extracted Kafka and navigate inside the Kafka directory. We will refer to this location as `kafka_home`. Note that we will be using the Linux executables in Kafka for the samples in this chapter. If you are on a windows environment the corresponding executables (alternatives for the `.sh` files) can be found in the `kafka_home/bin/windows` directory. For example, the alternative for the executable `bin/zookeeper-server-start.sh` for windows is `bin/windows/zookeeper-server-start.bat`.

Kafka requires a ZooKeeper (<https://zookeeper.apache.org/>) server to run. Apache ZooKeeper is a centralized service that provides various capabilities for managing and running distributed systems. Some of these capabilities include distributed synchronization, grouping services, naming services and so on. Figure 9.4 illustrates how ZooKeeper performs coordination of the Kafka cluster.

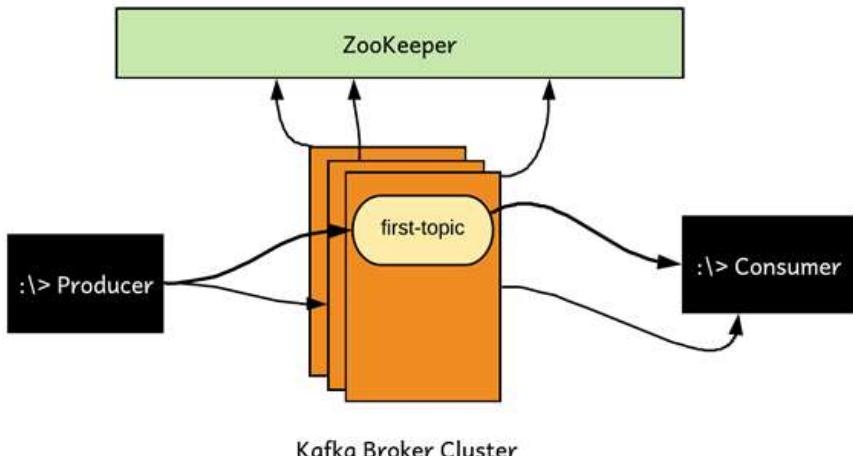


Figure 9.4 – ZooKeeper is used to perform coordination across the nodes in the Kafka cluster.

To run ZooKeeper, execute the following command using your command line terminal from within the `kafka_home` directory.

```
\> bin/zookeeper-server-start.sh config/zookeeper.properties
```

Once ZooKeeper is up and running, we can now start the Kafka server. Open a new tab on your command line client and navigate to the `kafka_home` directory and execute the command below.

```
\> bin/kafka-server-start.sh config/server.properties
```

Once the Kafka server is running, we can now create our topic for publishing messages into. Kafka by default comes with a utility tool that helps us to create topics easily. Open a new tab on your command line client and navigate to the `kafka_home` directory as before. Let's create our first topic by executing the command below. Note that we will use `firsttopic` as the name of the topic we are creating.

```
\> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --
partitions 1 --topic firsttopic
```

To see if the topic was created successfully you can execute the command below on your command line client. It should list the topic that we just created in the output.

```
\> bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

Now that we have Kafka up and running and a topic created on it, we can start to send and receive messages from Kafka. Open a new tab on your command line client from the `kafka_home` directory and execute the command below to start a console process that we can type message into.

```
\> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic firsttopic
```

This will start a prompt onto which you can type your messages. Before we start typing in any message let's start a message consumer process too so that we can observe the messages being typed in. Open a new terminal tab from the `kafka_home` directory and execute the command below to start a consumer process.

```
\> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic firsttopic --from-beginning
```

Now go back to the previous prompt and start typing in messages. Press the enter button after each message. You should notice that whatever messages you type in start appearing on the terminal tab on which you started the consumer process. What happens here is that your first command prompt delivers message to a topic named `firsttopic`. The consumer process is subscribed to this topic. Whenever a message is delivered to a topic the consumer process has subscribed to, it will receive the corresponding message. And in this particular case the consumer process simply outputs the received message to the console. This is illustrated in figure 9.5.

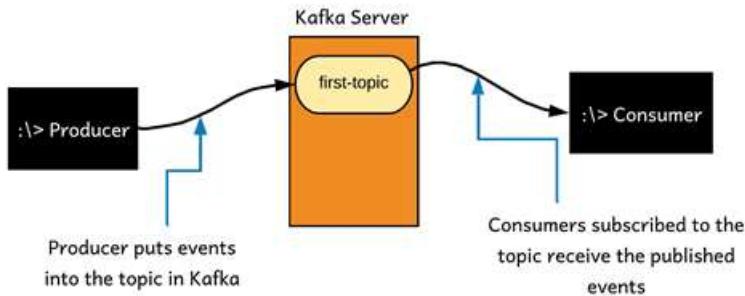


Figure 9.5 – The producer process puts events into the topic in the Kafka server. The consumer process receives the events being put.

Once you have tried out this part you can close off the console process that you typed messages into and the consumer process that displayed the messages being typed in. However please keep the Kafka processes up and running since we need Kafka for trying out the samples in the section 9.3.

9.3 Developing a microservice to push events to a topic in Kafka

In this section we discuss how we can develop a microservice in Spring Boot to push events to a topic in Kafka. The microservice receives messages via HTTP and those messages will then be converted into events and pushed to a topic on Kafka. You can find all the samples of this book in the <https://github.com/microservices-security-in-action/samples> GitHub repository. Navigate to the `chapter09/sample01` directory using your command line tool and execute the following command from within the `sample01` directory to build the Order Processing microservice.

```
\> mvn clean install
```

Make sure that you have Kafka running as per the instructions given in section 9.2. Execute the command below to run the Order Processing microservice.

```
\> mvn spring-boot:run
```

Once the Order Processing microservice has started you should see a topic named `ORDERS` being created on Kafka. You can verify it by listing the topics in Kafka by executing the command below in a new terminal window. You need to execute the command from within the `kafka_home` location.

```
\> bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

Let's now open up a new console process that prints the messages being sent to the `ORDERS` topic. As in section 9.2 we can execute the command below from within the `kafka_home` directory for this.

```
\> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic ORDERS --from-beginning
```

Once we have the console process running, we can now send a message to the Order Processing microservice and observe its behavior. Open a new terminal window and navigate to the `sample01` directory and execute the following cURL command to send a request to the Order Processing microservice to place an order.

```
\> curl http://localhost:8080/order -X POST -d @order.json -H "Content-Type: application/json" -v
```

The `order.json` file contains the message (in JSON format) that we sent to the Order Processing microservice. If the order has been placed successfully you should get an HTTP 201 response code to the request. If you observe the console process that prints messages on the `ORDERS` topic, you should see that the content within the `order.json` file (request payload) has been printed on the console. What happened here was that we used cURL to send an HTTP request to the Order Processing microservice to place an order. While the Order Processing microservice took care of the business logic related to placing the order, it also sent an event (message) to the `ORDERS` topic on Kafka. Any process that is subscribed to the `ORDERS` topic would now receive the order details via the topic and be able to execute their respective actions. In our case we just had a console process that prints the order details to the output. In

section 9.4 of the chapter we discuss about a process that listens to these order events and performs some actions. Figure 9.6 illustrates the scenario we tried out.

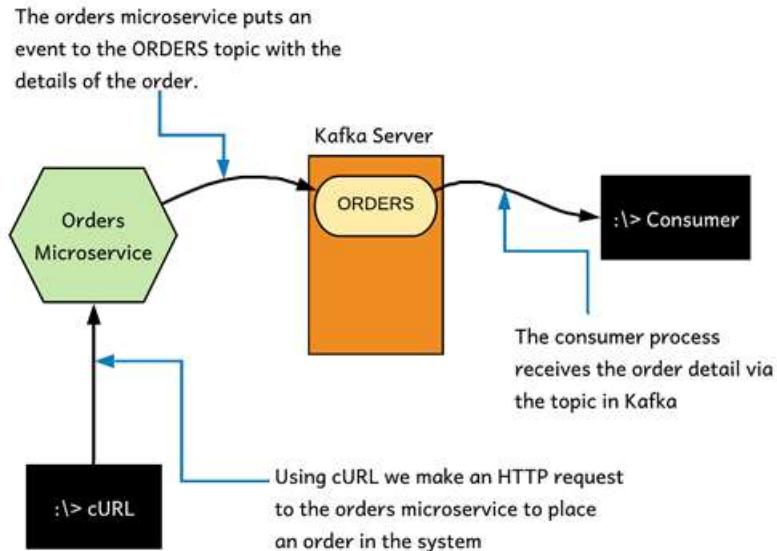


Figure 9.6 – cURL makes an HTTP request to the Order Processing microservice to place an order. After processing its logic, the Order Processing microservice put an event into the ORDERS topic in Kafka with the details of the order. The consumer process subscribed to the ORDERS topic would receive the details of the order through Kafka.

Let us take a look at the code that processed the order. The `OrderProcessing.java` class found in `sample01/src/main/java/com/manning/mss/ch09/sample01/service` is the class that hosts the microservice code. Its `createOrder` method is executed when it receives the HTTP request to place the order. Listing 9.2 lists this code.

Listing 9.2 – The createOrder method in the microservice

```

@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<?> createOrder(@RequestBody Order order) {

    if (order != null) {
        order.setOrderId(UUID.randomUUID().toString());
        URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
            .buildAndExpand(order.getId()).toUri();
        publisher.publish(order);      #A
        return ResponseEntity.created(location).build();
    }

    return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
}
    
```

#A Publishers the message into the Kafka topic after the order processing logic has been completed

The publish method used in Listing 9.2 is declared in the OrderPublisher.java class and listing 9.3 shows its code.

Listing 9.3 – Code that publishes the order to the Kafka topic.

```
@Service
public class OrderPublisher {

    @Autowired
    private Source source;

    public void publish(Order order){
        source.output().send(MessageBuilder.withPayload(order).build());      #A
    }
}
```

#A Sends the message to Kafka

We use Spring Cloud's Binders (<https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#spring-cloud-stream-overview-binders>) to bind our microservice to Kafka. Spring's binders' abstractions allow us to bind our program to different types of streaming implementations. To find out how it exactly connects to the Kafka server take a look at the applications.properties file located at chapter09/sample01/src/main/resources directory. You will see several properties in there. One property tells our program which topic in Kafka to connect to (ORDERS) and there are two properties, which specify the zookeeper connection details (in this case localhost). You may notice that we haven't specified the zookeeper ports. This is because we are working with the default ports in zookeeper (2181). In case you want to use non-default ports, you have to specify the connection details in the format of <host>:<port> (localhost:2181).

9.4 Developing a microservice to read events from a Kafka topic

In this section we discuss how we can create a microservice that reads and acts upon the events received from our Order Processing microservice. In section 9.3 we discussed how we could get the Order Processing microservice to send events to a topic in Kafka whenever an order was processed. In this section we discuss how we can implement our Buying History microservice, which tracks a customer's buying patterns.

The microservice that tracks a customer's buying patterns listens on the ORDERS topic in Kafka. Whenever it receives an Order event it will act upon the event and execute its logic related to analyzing and recording the purchase. Figure 9.7 illustrates this architecture.

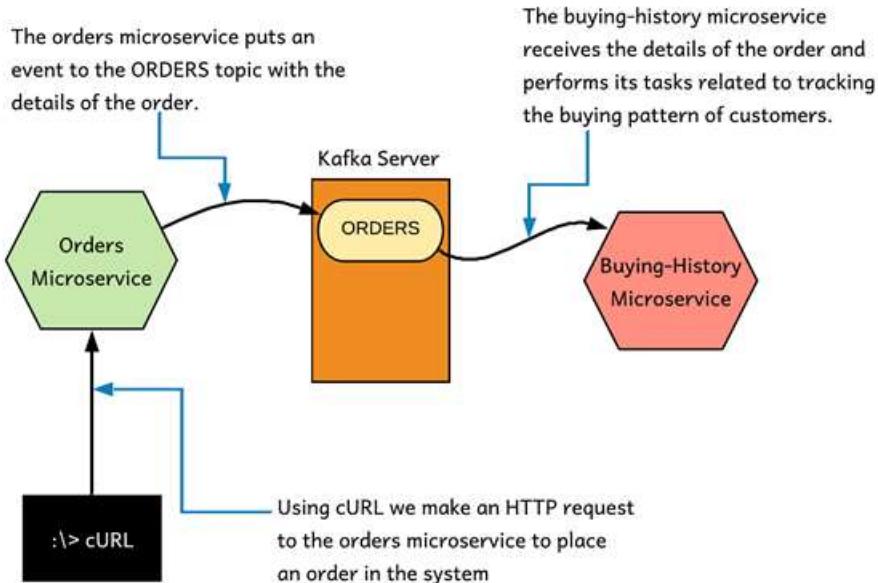


Figure 9.7 – The Buying History microservice now receives the order details via Kafka. It will then start processing its task related to tracking the buying patterns of customers. This task is done asynchronously to the processing of the order.

Let's first run the sample of this section to see how this works. You can find the code related this section inside the `chapter09/sample02` directory. You need to have the Kafka server up and running from section 9.2 and the Order Processing microservice up and running from `sample01`. The Order Processing microservice is the event source to the Kafka topic. The Buying History microservice is the one, which consumes the events that are received on the Kafka topic. Open up your command line tool and navigate to the `sample02` directory and execute the following command to build the sample.

```
\> mvn clean install
```

Once the sample is built, execute the command below to start the Buying History microservice.

```
\> mvn spring-boot:run
```

When the Buying History microservice starts it will subscribe to the Kafka topic named `ORDERS`. Once the Buying History microservice is running you can now send an HTTP request to the Order Processing microservice to create an order. Open a new tab on your command line tool and navigate to the `sample02` directory and execute the command below to make the orders request. Note the Order Processing microservice from section 9.3 handles this request.

```
\> curl http://localhost:8080/order -X POST -d @order.json -H "Content-Type: application/json" -v
```

If successful, this command should respond with a 201 response code. If you observe the console output of the terminal tab that runs the Buying History microservice you should see the following message.

```
Updated buying history of customer with order: <order_id>
```

The Buying History microservice received the order details via the Kafka topic ORDERS. Upon receiving this event it will now execute its logic to track the purchase history of the customer. In this particular case it prints a message to the console saying it received the order event and processed it. Figure 9.8 illustrates the sequence of events that happened.

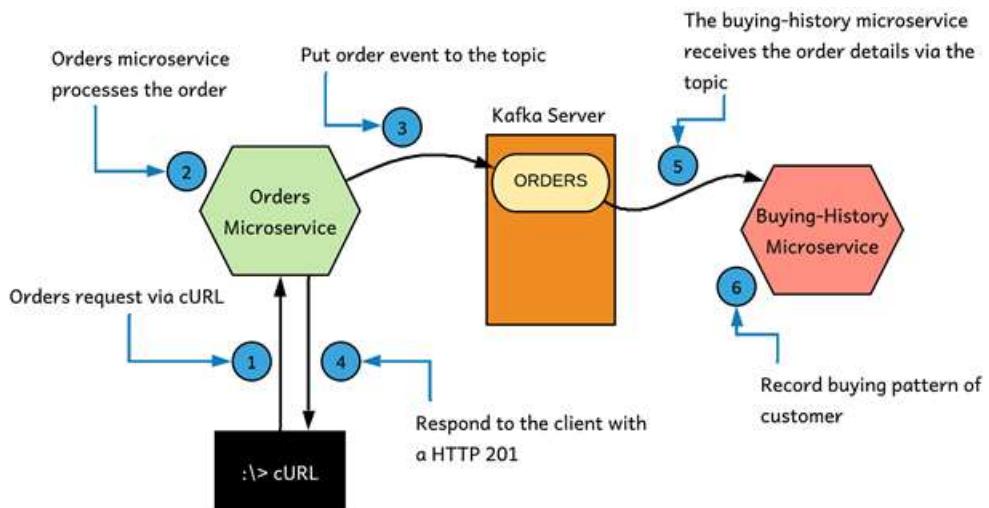


Figure 9.8 – The sequence of events that happen when a client (cURL) makes a request to place an order. Note that steps 4 and 5 may happen parallelly since they are on two independent processes.

Let's take a look at the code of the Buying History microservice, which received the message from Kafka when the order was placed. The `BuyingHistoryMicroservice.java` class located at `sample02/src/main/java/com/manning/mss/ch09/sample02` contains the code of the Buying History microservice. Listing 9.4 lists out this code.

Listing 9.4 – Microservice code that reads messages from Kafka

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class BuyingHistoryMicroservice {

    public static void main(String[] args) {
        SpringApplication.run(BuyingHistoryMicroservice.class, args);
    }

    @StreamListener(Sink.INPUT)      #A
}
```

```

public void updateHistory(Order order) {
    System.out.println("Updated buying history of customer with order: " +
order.getOrderId());
}
}

```

#A This annotation tells the Spring runtime to trigger this method whenever a message is received on the topic it is listening to

As we discussed in section 9.3 the Spring runtime is configured to connect to Kafka through the properties defined in the applications.properties file located at sample02/src/main/resources directory.

9.5 Using Transport Layer Security (TLS) to protect data in transit

In this section we discuss how to enable transport layer security (TLS) to protect the data that is being passed to and from the Kafka server to the Kafka producers and consumers (microservices). In sections 9.3 and 9.4 we implemented the Order Processing microservice, which sent events to Kafka and the Buying History microservice, which received events from Kafka. In both these cases the data that was passed along the wire between the Kafka server and the respective microservices was in plaintext. Which means that anyone having access to the network layer of the microservices deployment could read the messages being passed between the two microservices and the Kafka server. This is not ideal. We should encrypt the data being passed via the network and the way to do that is to enable TLS between the communicating parties. Although the microservice itself is not configured with TLS, we are not focusing on enabling TLS for the microservice in this chapter since we already covered that in chapter 6. Let's look at how to enable TLS on Kafka.

9.5.1 Creating and signing the TLS keys and certificates for Kafka

In this section we discuss how to create a key and a certificate for the Kafka server to enable TLS communication. We also discuss how to sign the Kafka certificate using a self-generated certificate authority (CA). To create the CA and other related keys you can use the gen-key.sh file inside chapter09/keys directory, which includes a set of OpenSSL commands. OpenSSL is a commercial-grade toolkit and cryptographic library for TLS, available for multiple platforms. Here we run OpenSSL as a Docker container, so you do not need to worry about installing OpenSSL in your computer, but you should have Docker installed. If you are new to Docker, please check appendix A, but you do not need to be thorough with Docker to follow rest of this section. To spin up the OpenSSL Docker container, run the following docker run command from the chapter09/keys directory (listing 9.5).

Listing 9.5. Spins up OpenSSL in a Docker container

```

\> docker run -it $(pwd):/export prabath/openssl
#

```

The above `docker run` command starts OpenSSL in a Docker container, with a volume mount, which maps `keys` (or the current directory, which is indicated by `$(pwd)`) directory from the host file system to the `/export` directory of the container file system. This volume mount helps you to share part of the host file system with the container file system. When the OpenSSL container generates certificates, those are written to the `/export` directory of the container file system. Since we have a volume mount, everything inside the `/export` directory of the container file system is also accessible from the `keys` directory of the host file system. When you run the command in listing 6.1 for the first time, it may take couple of minutes to execute and ends with a command prompt, where we can execute our script to create all the keys. The following command, which runs from the Docker container, executes the `gen-key.sh` file, which is inside the `export` directory of the container, and its is the same script, which is inside the `keys` directory of your local file system.

```
# sh /export/gen-key.sh
```

Now, if you look at the `keys` directory in the host file system, you will find a set of files as shown in listing 9.6. If you want to understand, what happens underneath the script, please check appendix K.

Listing 9.6. Generated keys and certificates

```

└── buyinghistory
    ├── buyinghistory.jks      #A
    └── truststore.jks        #B
└── ca
    ├── ca_cert.pem      #C
    └── ca_key.pem       #D
└── kafka_server
    ├── kafka_server.jks    #E
    └── truststore.jks      #F
└── orderprocessing
    ├── orderprocessing.jks  #G
    └── truststore.jks      #H

```

#A Contains the private key and the CA signed certificate for Buying History microservice

#B Contains the public certificate of the CA

#C The public certificate of the CA

#D The private key of the CA

#E Contains the private key and the CA signed certificate for Kafka server

#F Contains the public certificate of the CA

#G Contains the private key and the CA signed certificate for Order Processing microservice

#H Contains the public certificate of the CA

9.5.2 Configuring TLS on the Kafka server

In this section we discuss how to enable TLS on the Kafka server. This requires changing a few configuration parameters and restarting the Kafka server. To enable TLS on Kafka, first make sure the Kafka server is shutdown if already running (but let the Zookeeper server running from section 9.2). You need to press the CTRL + C (Control and C) buttons on your keyboard on the respective command line terminal processes. After the process has been shut down, use

your command line client tool or file explorer to navigate to `kafka_home/config` directory. Open `server.properties` file using your text editor of choice (see listing 9.7) and add the following properties to the file.

Listing 9.7. Content of server.properties file

```
listeners=PLAINTEXT://:9092,SSL://:9093 #A
ssl.keystore.location=chapter09/keys/kafka_server/kafka_server.jks #B
ssl.keystore.password=manning123
ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1
ssl.keystore.type=JKS
ssl.secure.random.implementation=SHA1PRNG
```

#A The listeners configuration tells the Kafka server to be listening on ports 9092 for plaintext (non secure) connections and port 9093 for secure connections.

#B Provide the absolute path for the `kafka_server.jks`

Once the above configurations are done you can save and close the file and start the Kafka server. Use your command line client and navigate to `kafka_home` and execute the command below to start Kafka.

```
\> bin/kafka-server-start.sh config/server.properties
```

9.5.3 Configuring TLS on the microservices

In this section we discuss how to enable TLS on our Order Processing and Buying History microservices. We need to enable a few properties and provide these microservices with the details of the `truststore.jks`, which we created in section 9.5.1. You can find the updated code of the Order Processing microservice inside the `chapter09/sample03/orders_ms` directory and the updated code of the Buying History microservice inside the `chapter09/sample03/buying_history_ms` directory.

First, we need to configure the keystores, which we created in section 9.5.1 with the Order Processing and Buying History microservices. Lets copy the keystore files from `chapter09/keys` directory to the corresponding microservice, by executing the following commands from the `chapter09` directory.

```
\> cp keys/orderprocessing/*.jks sample03/orders_ms/
\> cp keys/buyinghistory/*.jks sample03/buying_history_ms/
```

When configuring keystores with the Order Processing and Buying History microservices we need to worry about two things.

- Enable HTTPS for the Order Processing microservice. So the client applications, which talk to the Order Processing microservice, must use HTTPS. We do not need to do the same for the Buying History microservice, because we do not expose it over HTTP.
- Configure both Order Processing and Buying History microservices to trust the public certificate of the certificate authority, which signed the public certificate of the Kafka server. The Order Processing microservice connects to the Kafka server to publish messages and the Buying History microservice connects to the Kafka server to read

messages. Both of these communications now happen over TLS and both the microservices have to trust the public certificate of the Kafka server.

Use your file editor of choice and open the application.properties file located in orders_ms/src/main/resources and provide proper values for the spring.kafka.ssl.trust-store-location and spring.kafka.ssl.trust-store-password properties. These are the properties that instruct Spring Boot to trust the public certificate of the Kafka server. Also check the values of server.ssl.key-store and server.ssl.key-store-password properties. These two properties enable HTTPS transport for the Order Processing microservice. Save and close the file once these changes are done. If you go with default values, we used in our samples, you do not need do any changes. Do the same for the application.properties located at bying_history_ms/src/main/resources and save and close the file. Once both these changes are done use your command line client and navigate to the orders_ms directory and execute the following command to build the new Order Processing microservice.

```
\> mvn clean install
```

Once the build is successful execute the following command to run the Order Processing microservice. Make sure you have the Kafka server running before you execute this command. In case you are still running the Order Processing microservice from sample01 (section 9.3 of this chapter) make sure it has been shut down as well.

```
\> mvn spring-boot:run
```

Let us now do the same for our Buying History microservice. In a new terminal tab navigate to the bying_history_ms directory and execute the command below to build the Buying History microservice.

```
\> mvn clean install
```

Once the build is successful execute the command below to run the Buying History microservice. In case you are still running the Buying History microservice from sample02 (section 9.4 of this chapter) make sure it has been shut down as well.

```
\> mvn spring-boot:run
```

Similar to that of section 9.4, we can now make an order request using cURL as below. Open a new terminal tab on your command line client and navigate to the orders_ms directory and execute the following command.

```
\> curl -k https://localhost:8080/order -X POST -d @order.json -H "Content-Type: application/json" -v
```

If you observe the console output of the terminal tab on which you ran the buying-history microservice you should see an output as below.

```
Updated buying history of customer with order: <order_id>
```

Let's take a look at the configuration changes we made to the Order Processing microservice to make it work on TLS. Open the application.properties file located at orders_ms/src/main/resources using a text editor or your IDE. You should observe the properties shown in listing 9.8 and the code annotations explain what each of those properties means.

Listing 9.8. Content of application.properties file

```
server.ssl.key-store: orderprocessing.jks      #A
server.ssl.key-store-password: manning123      #B
spring.kafka.bootstrap-servers=localhost:9093   #C
spring.kafka.properties.security.protocol=SSL  #D
spring.kafka.properties.ssl.endpoint.identification.algorithm=    #E
spring.kafka.ssl.trust-store-location=file:truststore.jks      #F
spring.kafka.ssl.trust-store-password=manning123      #G
spring.kafka.ssl.trust-store-type=JKS            #H
```

#A The location of the keystore, which carries keys to enable HTTPS communication

#B The password of the keystore

#C instructs the microservice to connect to the TLS port (9093) of Kafka. If no port specified then the microservice connects to the default Kafka port 9092, which is the plain-text port (no TLS)

#D Sets the protocol for transport layer security

#E By setting this empty we effectively get our microservice to ignore the hostname verification of the server certificate. In a production deployment, you should not do this.

#F The location of the keystore, which carries CA's public certificate

#G The password of the trust store

#H The type of the trust store

NOTE: By setting the value `ssl.endpoint.identification.algorithm` property as empty in listing 9.8 we effectively get our microservice to ignore the hostname verification of the server certificate. On a real production system, we wouldn't ideally do this. A production server would have a proper DNS setup for the Kafka server, and it would be exposed using a valid hostname (not localhost). We would use the hostname in our server certificate and enforce hostname verification on the client (microservice). But for now, since we do not have a proper hostname to use, we will skip the hostname verification on the client.

The application.properties file on the Buying History microservice also contains the same configurations as the Order Processing microservice. You can observe those configurations by looking at the file located at `buying_history_ms/src/main/resources`.

9.6 Using mutual Transport Layer Security (mTLS) for authentication

In section 9.5 we looked at enabling TLS between the microservices and the Kafka server. In this section we discuss how to enable mutual TLS between the Order Processing / Buying History microservices and the Kafka server for client authentication. By enabling TLS between the microservices and Kafka we ensured the data being transmitted over the network between these parties were encrypted. Nobody sniffing into the network would be able to read the data being transmitted unless they had access to the server's (Kafka) private key. By enabling TLS, we also made sure that the microservices (clients) are connected to the intended/trusted Kafka

server and not to anyone pretending to be it. One problem that still remains however is that anyone having network level access to the Kafka server and trusts the CA that signed Kafka's certificate would be able to publish events to the Kafka server and receive events. For example, we would not want anyone impersonating the Order Processing microservice and sending bogus order events to Kafka. Figure 9.9 illustrates this scenario.

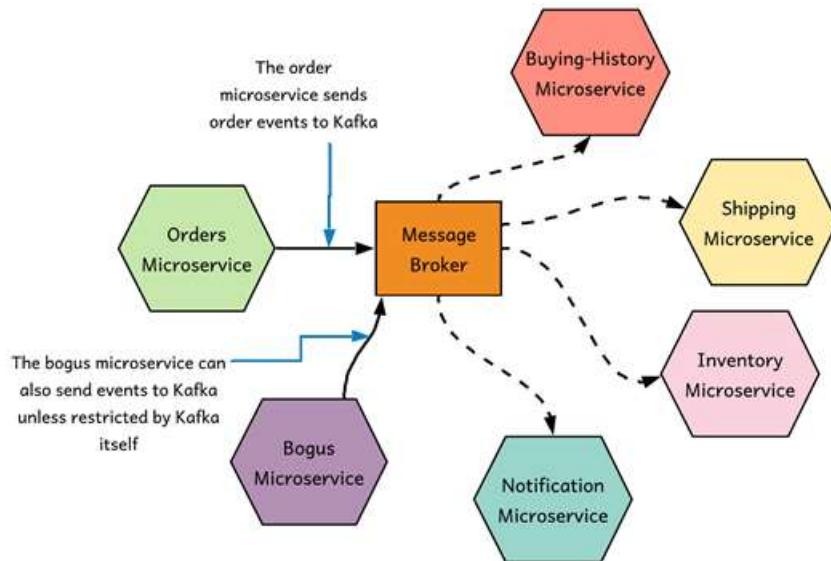


Figure 9.9 – The bogus microservice impersonates the Order Processing microservice by sending order events to Kafka. This makes the other microservices process these order events. Kafka needs to explicitly allow only the trusted microservices to connect to it.

As shown in figure 9.8 the bogus Order Processing microservice is also sending events to Kafka, which would trigger false events in the system and make the other microservices act upon it. This would cause our system to break. To prevent this from happening we need to make sure that it is only the trusted Order Processing microservice (and other trusted microservices) that are permitted to send and receive events from Kafka. With TLS, in section 9.5 we established this trust one way where the client was made to trust the server before sending events to it. We now need to enable trust both ways where the server also trusts the client's connecting to it through mutual TLS. Let's see how to enable mutual TLS on Kafka.

Make sure your Kafka server has been stopped. You can stop it by pressing the **ctrl + c** (control + c) button combinations on the terminal process that is running the Kafka server process. Then open the `server.properties` file located at `kafka_home/config` with your text editor. Add the following two configuration properties to it. Make sure to provide the absolute path pointing to the `chapter09/keys/kafka_server/truststore.jks`, as the

value of the `ssl.truststore.location` property. We created `truststore.jks` in section 9.5.1.

```
ssl.truststore.location=chapter09/keys/kafka_server/truststore.jks
ssl.truststore.password=manning123
```

Change the value of the property `ssl.client.auth` to `required` as shown below (in the same `server.properties` file). This tells the Kafka server to authenticate the clients connecting to Kafka before allowing connection to be established.

```
ssl.client.auth=required
```

Once these properties are set, save and close the file and start the Kafka server. The Kafka server is now ready to accept requests from clients who can authenticate themselves. If you now attempt to run the microservices from section 9.5 they would now start failing since we have mandated client authentication on Kafka. The microservices from section 9.5 did not have the mutual TLS configurations enabled, hence the failure. The next step would be to configure Order Processing and Buying History microservices to be able to authenticate themselves when connecting to the Kafka server. You can take a look at the code related to the microservices from the `chapter09/sample03` directory.

Let's see how to enable TLS mutual authentication client support for the Order Processing microservice. Open the `application.properties` file located in `chapter09/sample03/orders_ms/src/main/resources`. You should see three new properties being added compared to the same in listing 9.8. Those are commented out by default, so please uncomment them. The listing 9.9 shows the updated configuration.

Listing 9.9. Content of application.properties file with mTLS support

```
spring.kafka.ssl.key-store-location=file:orderprocessing.jks      #A
spring.kafka.ssl.key-store-password=manning123      #B
spring.kafka.ssl.key-store-type=JKS      #C
```

#F The location of the keystore, which carries microservice's public certificate and the private key. We created this keystore in section 9.5.1.

#G The password of the keystore

#H The type of the keystore

Do the same for the Buying History microservice as well. You can find the `application.properties` file corresponding to the Buying History microservice located in `chapter09/sample03/buying_history_ms/src/main/resources`. There we have to use `buyinghistory.jks` as the location of the keystore file. We created the `buyinghistory.jks` file in section 9.5.1 and copied to the `chapter09/sample03/buying_history_ms` directory in section 9.5.3.

Once both the `application.properties` files are updated, we can now build and run the two microservices. Make sure you are not running the microservices from previous sections before you execute the next steps. Build the Order Processing microservice by executing the

following command using your command line client within the sample03/orders_ms directory.

```
\> mvn clean install
```

Once built execute the following command to run the Order Processing microservice.

```
\> mvn spring-boot:run
```

Perform the same steps to build and run the Buying History microservice from the sample03/buying_history_ms directory and then execute the following cURL command from sample03/ orders_ms to make an order request to the Order Processing microservice.

```
\> curl -k https://localhost:8080/order -X POST -d @order.json -H "Content-Type: application/json" -v
```

You should receive a successful response from this command, and you should notice the following message being printed on the terminal tab that is running the Buying History microservice.

```
Updated buying history of customer with order: <order_id>
```

9.7 Controlling access to Kafka topics with ACLs

In this section we discuss how we can control access to topics in Kafka by client (microservice). In section 9.6 we discussed how to enable client authentication using mutual TLS (mTLS). We discuss some of the use cases we need to address when it comes to controlling who has permissions to read and write from topics in Kafka and discuss how we can achieve those use cases using access control lists (ACLs).

In section 9.6 we discussed how we could control connections to the Kafka server using mutual TLS. We made sure that only trusted clients could connect to Kafka. We used mutually trusted certificates both on the client (microservice) and the Kafka server to achieve this. We did that by getting a mutually trusted certificate authority (CA) to sign the certificates of each other (client and server). We now want to get to a state where only selected microservices are given selective permissions to Kafka topics. For example, we need to ensure that only the Order Processing microservice can publish events into the ORDERS topic in Kafka. The Buying History microservice should only be permitted to read events from the ORDERS topic. We cannot achieve this with mutual TLS only. The Buying History microservice from section 9.6, since granted connection rights through mutual TLS, can technically even publish events to the ORDERS topic even though the code samples we used did not. Figure 9.10 illustrates this scenario.

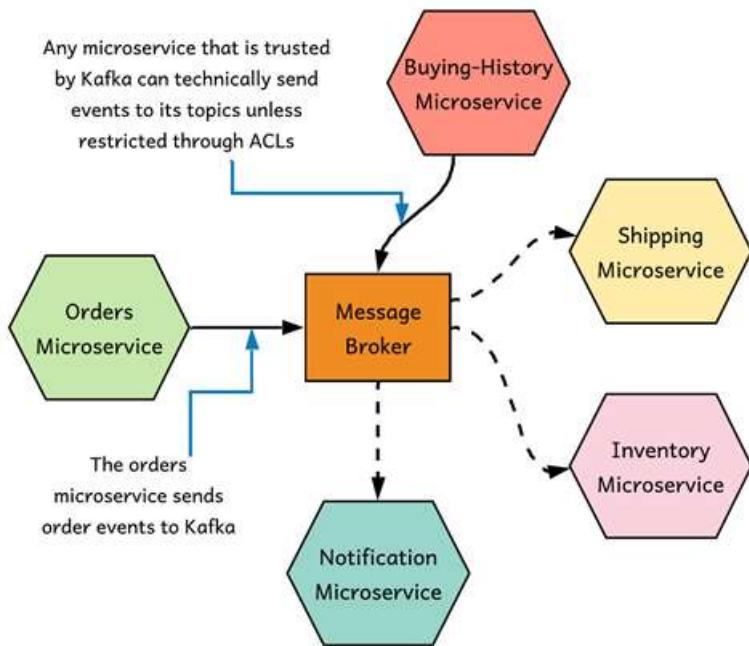


Figure 9.10 – The buying-history microservice sends events to Kafka topics. Any microservice that is trusted by Kafka can technically send events to its topics unless restricted by ACLs. These events are delivered to microservices that are subscribed to the Kafka topic except if they have been restricted by ACLs.

Let us take a look at how we can prevent this from happening. What we have so far achieved in this chapter is client and server authentication. To enforce more fine-grained access control on Kafka topics, we need to implement authorization on Kafka. Kafka provides a way of implementing authorization, using Access Control Lists (ACLs). An ACL is basically a rule on Kafka, which either permits or denies a particular entity from performing a particular action on a Kafka resource. Kafka ACLs are defined in the following format.

Principal P is [Allowed/Denied] **Operation O** From **Host H** on any **Resource R** matching **ResourcePattern RP**

The **Principal** represents the entity who/which attempts to perform an operation (the client). An **Operation** could be any one of things such as creating a topic, writing events to a topic, reading events from a topic and so on. A **Host** is the IP address of the Kafka client. A **Resource** is an entity on Kafka such as a topic, a consumer group and so on. A **ResourcePattern** is a pattern that is used to identify the resources on which the rule is to be applied on.

9.7.1 Enabling ACLs on Kafka and identifying the clients

In this section we discuss how we can enable ACLs on Kafka through configuration. First make sure your Kafka server is shut down before you make these changes to the configuration file. Note that we are continuing our configurations from the previous sections. In case you have directly jumped on to this section you need to first enable mutual TLS on Kafka by reading through sections 9.5 and 9.6. Open the `server.properties` file located at `kafka_home/config` with your text editor and add the following properties to it.

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
allow.everyone.if.no.acl.found=true
ssl.principal.mapping.rules=RULE:^CN=(.*?)$/$1/L,DEFAULT
```

The `authorizer.class.name` property contains the name of the class that executes the authorization logic in Kafka. In this case we use the default class that ships with Kafka itself. If required, you can override the default implementation and add your custom authorization class name to the configuration file. By adding this property to the configuration file, we enable ACLs on Kafka. The `allow.everyone.if.no.acl.found` property specifies the action to be performed in case an ACL is not found for a given resource. By default, Kafka denies access to resources (topics) if no ACL are declared on it. By setting this to true we override that behavior by saying unless a rule is applied on a resource via an ACL let anyone access it without restriction.

The `ssl.principal.mapping.rules` property defines the patterns to identify the principal (Kafka client). Since we use mutual TLS to authenticate clients (microservices), the Kafka server identifies a client through the properties in the corresponding client certificate. We can have multiple rules defined under the `ssl.principal.mapping.rules` property, and each rule is defined using the following format.

```
RULE:pattern/replacement/[LU]
```

This pattern is a regular expression to identify the principal from the provided credentials (for example, a client certificate). By default, when we use an X509 certificate for authentication (or mTLS), the principal identifier is its distinguished name or the DN. The distinguished name of a certificate is a string that combines the certificates details as key values pairs delimited by commas. For a certificate having the CN (common name) as `orders.ecomm.com` the distinguished name looks like the following.

```
CN=orders.ecomm.com,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown
```

Unless the `ssl.principal.mapping.rules` property is specified, Kafka would check for the full distinguished name as shown above, when applying its ACLs. This property helps overriding the default behavior, for example with a principal mapping rule such as `RULE:^CN=(.*?),OU=(.*?),O=(.*?),L=(.*?),ST=(.*?),C=(.*?) $/$1/L` we effectively instruct Kafka to accept any certificate (through the regular expression) and consider only the value of its CN (\$1) in lowercase (L) when matching against declared ACLs. We do that by setting the replacement string to \$1 and by setting the optional case matcher

to L to indicate lower-case. For a client (microservice), which has a certificate whose distinguished name matches CN=orders.ecomm.com, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown, Kafka identifies the client as orders.ecomm.com and we need to set ACLs against that name. When we created certificates for the Order Processing and Buying History microservices in section 9.5.1, we used orders.ecomm.com and bh.ecomm.com as the CN respectively for each microservice.

Once the above configurations are done, save and close the server.properties file and restart the Kafka. Now we have enabled ACLs on Kafka the next step is to define the ACLs on it.

9.7.2 Defining ACLs on Kafka

In this section we use our command line tool to define ACLs on Kafka. Since we now have the ACLs enabled on Kafka and the server is up and running, we can proceed to define the rules we need to apply on the relevant topics. We will setup two rules.

- Allow only the Order Processing microservice to publish events to the ORDERS topic, or be the producer of the ORDERS topic.
- Allow only the Buying History microservice to consume events from the ORDERS topic or be a consumer of the ORDERS topic.

Open your command line tool and navigate to the kafka_home directory and execute the following command.

```
\> bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:"orders.ecomm.com" --producer --topic ORDERS
```

This will show you an output similar to the following.

```
Adding ACLs for resource `Topic:LITERAL:ORDERS`:
User:orders.ecomm.com has Allow permission for operations: Write from hosts: *
User:orders.ecomm.com has Allow permission for operations: Describe from hosts: *
User:orders.ecomm.com has Allow permission for operations: Create from hosts: *
```

As you can observe from the output, we now have an ACL, which says the user orders.ecomm.com is allowed to create, write-to and describe the ORDERS topic. By create it implies that orders.ecomm.com is allowed to create the topic if it doesn't exist. By write it implies that orders.ecomm.com is allowed to publish events to the topic and by describe it implies that orders.ecomm.com is allowed to view the details of the topic. These operations are allowed from any hosts since we have not explicitly mentioned which hosts to restrict these operations to. In case you want to remove the ACL we just created, you can use the following command, but please do not run it till we complete this chapter.

```
\> bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --remove --allow-principal User:"orders.ecomm.com" --producer --topic ORDERS
```

Next execute the following command to enable read access to the ORDERS topic for our Buying History microservice, having a certificate with the CN bh.ecomm.com.

```
\> bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:"bh.ecomm.com" --operation Read --topic ORDERS
```

This will show you can output similar to the following.

```
Adding ACLs for resource `Topic:LITERAL:ORDERS`:  
User:buyinghistorysms has Allow permission for operations: Read from hosts: *
```

In case you want to remove the ACL we just created, you can use the following command, but please do not run it till we complete this chapter.

```
\> bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --remove --allow-principal User:"bh.ecomm.com" --operation Read --topic ORDERS
```

You can list the ACLs on the ORDERS topic by executing the following command.

```
\> bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --list --topic ORDERS
```

This will show you all the ACLs applied on the ORDERS topic and you should see an output similar to the following.

```
Current ACLs for resource `Topic:LITERAL:ORDERS`:  
User:orders.ecomm.com has Allow permission for operations: Write from hosts: *  
User:orders.ecomm.com has Allow permission for operations: Describe from hosts: *  
User:orders.ecomm.com has Allow permission for operations: Create from hosts: *  
User:bh.ecomm.com has Allow permission for operations: Read from hosts: *
```

We now have all the ACLs in place that control access to the ORDERS topic. Assuming that both the Order Processing and Buying History microservices are running from the section 9.6, first stop both and restart. This is required as we restarted the Kafka server. To test how ACLs work with Kafka, run the following cURL command from `chapter09/ sample03/orders_ms` directory to make an order request to the Order Processing microservice.

```
\> curl -k https://localhost:8080/order -X POST -d @order.json -H "Content-Type: application/json" -v
```

You should receive a successful response from this command, and notice the following message being printed on the terminal tab that is running the Buying History microservice.

```
Updated buying history of customer with order: <order_id>
```

9.8 Summary

- In a typical microservices deployment, the microservice, which accepts requests from a client application, becomes a trigger for the rest of the microservices involved in the flow to be executed. This microservice may trigger both synchronous and asynchronous types of events to other microservices to trigger their execution.
- Reactive programming is a declarative programming paradigm that relies on data streams and the propagation of change events in a system.
- A microservice being directly connected to another microservice causes scalability and

maintenance inefficiencies due to its dependency hierarchy. For asynchronous events, we can build reactive microservices, which are capable of reacting to events that occur in a system, which reduces dependencies among microservices.

- Apache Kafka is a distributed stream platform that operates similar to message queues.
- In a microservices deployment, we use Apache Kafka as a message broker to deliver events from source microservices to target microservices asynchronously.
- With the reactive pattern, we eliminate direct links between microservices and increase the operational efficiencies. It further reduces the response times of the microservices for the users and provides a better foundation for introducing new functionality into the system.
- Transport Level Security (TLS) is used between the microservices and Kafka to encrypt messages being transferred between the two.
- Kafka uses mutual TLS (mTLS) to control which microservices are permitted to connect to it or to authenticate the clients connecting to it.
- Kafka uses access control lists (ACLs) as an authorization mechanism, which either permits or denies various microservices performing different types of actions on Kafka resources such as topics.

10

Conquering container security with Docker

This chapter covers

- Securing service to service communication with JWT and mutual transport layer security (mTLS) in a containerized environment
- Managing secrets in a containerized environment
- Signing and verifying Docker images with Docker Content Trust (DCT)
- Running Docker Bench for security

The benefits of microservices architecture come at a cost. Unless you have proper infrastructure to support microservices development and deployment with a continuous integration/continuous delivery (CI/CD) pipeline, chances are high that you'll fail to meet your objectives. Let us reiterate, one key objective of microservices architecture is the speed to production. With hundreds of microservices, management becomes a nightmare, unless you have the right tools for automation. Packaging, distribution and testing of microservices in various environments, before getting into production, in an efficient, less error prone way is important. Over time Docker has become the most popular tool (or the platform) for packaging and distributing microservices. It provides an abstraction over the physical machine. Docker simply does not only package your software, but all its dependencies too. In this chapter we discuss deploying and securing microservices in a containerized environment with Docker. We cover basic constructs of Docker in appendix A; so if you are new to Docker, please follow appendix A first. Even if you are familiar with Docker, we still recommend you at least skim through appendix A. The rest of the chapter assumes you have the knowledge of appendix A.

In practice, you don't have only Docker; Docker is used within a container orchestration framework. A container orchestration framework (such as Kubernetes or Docker Swarm) is an abstraction over the network. Container orchestration software like Kubernetes lets you deploy, manage and scale containers in a highly distributed environment with thousands of nodes, or even more. In chapter 11 we discuss the role of a container orchestration framework (Kubernetes) in securing microservices. We wanted to highlight this at the beginning of this chapter, to set your expectations right. The security of a microservices deployment should be thought of under the context of a container orchestration framework, not just about container security in isolation. Also an important pattern we see in a containerized deployment is the *service mesh* pattern. At a very high-level the *service mesh* pattern deals with service-to-service communication, and helps taking out most of the burden from microservices and delegate security processing to a proxy. Istio is the most popular service mesh implementation and in chapter 12 we discuss securing microservices in a containerized environment with Istio. Once again, this is another reason, why we should not think about container security in isolation, in securing microservices.

10.1 Running Security Token Service on Docker

In chapter 7 we learnt about JSON Web Tokens (JWT) and how to secure microservices with a JWT issued by a Security Token Service (STS). This use case from chapter 7 is illustrated in figure 10.1. In this section we build the same use case, but deploy STS on Docker.

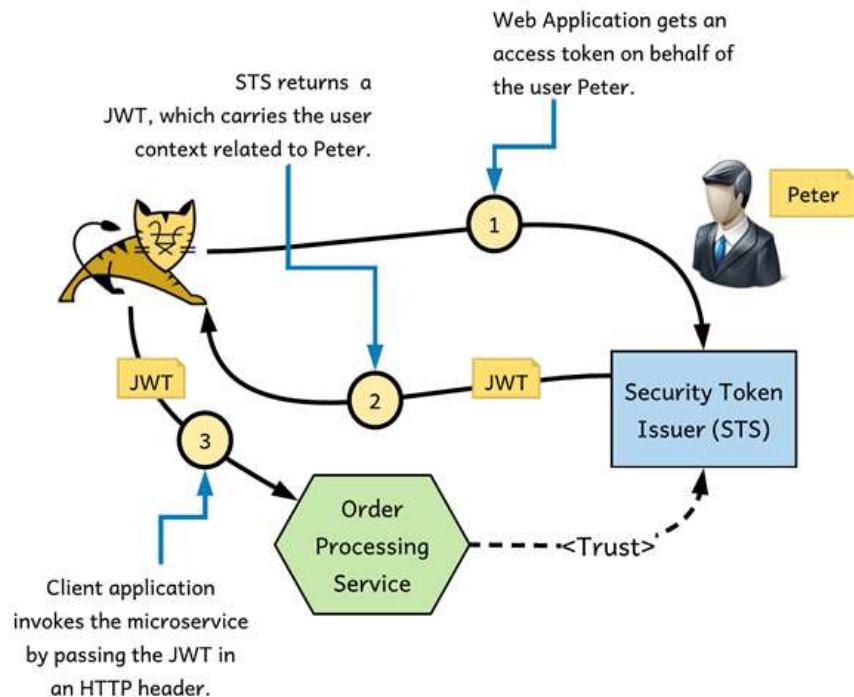


Figure 10.1 The STS issues a JWT (self-contained access token) to the web application (probably following OAuth 2.0) and the web application uses the token to access the Order Processing microservice on behalf of Peter.

The source code related to all the samples in this chapter is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, inside the chapter10 directory. The source code of the STS, which is a Spring Boot application developed in Java, is available in the chapter10/sample01 directory. To build the STS project, and create a Docker image, execute the following commands (listing 10.1), from the chapter10/sample01 directory. These are standard commands that you'd be using in any Docker project and the section A.4 of appendix A, explains them in details, in case you are not familiar with Docker.

Listing 10.1. The commands to build the STS and create a Docker image

```
\> mvn clean install
\> docker build -t com.manning.mss.ch10.sample01:v1 .
\> docker run -p 8443:8443 com.manning.mss.ch10.sample01:v1
```

Now let's test the security token service, with the following cURL command. This is exactly the same cURL command we used in chapter 7, under the section 7.6.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
```

```
application/x-www-form-urlencoded; charset=UTF-8" -k -d
"grant_type=password&username=peter&password=peter123&scope=foo"
https://localhost:8443/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret (which are hard-coded in to the STS). If everything works fine, the security token service returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiE1NTEzMTIzMzYsInVzZXJfbmFtZSI6InBldGVyIiwiXXV0aGyaXRpZXMiOls1Uk9MRV9VU0VSI10sImp0aSI6jRkMmjnJQ4LTQ2MWQtNGV1Yy1hZT1jLTW1YUWxZjA4ZTJhMiIsImNsawVudF9pZC16ImFwcGxpY2F0aW9uaWQiLCJzY29wZSI6WyJmb28iXX0.tr4yUmGLtsH7q9Ge2i7gxyTs0Oa0RS0Yoc2uBuAW50VKZcVsIIW3bDN0FVBzimpAPy33tvicFR0hBFoVThqKXzzG00SkURN5bnQ4uFLAP0NpZ6BuDjvVmwxNxrQp21Vx141Q4eTvuyZozjUSCxzC1LNw5EFFi22J73g1_mRm2jdEhBp1TvMaRKLBdk2hzIDVKzu5oj_g0DBFm3a1s-IJjYoCimIm2igcesXkhipRjtjNcrJSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHzPM8yz_K97j1T6Rr1XZB1veyjfKs_XIXgU5qizRm9mt5xg", "token_type": "bearer", "refresh_token": "", "expires_in": 5999, "scope": "foo", "jti": "4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

10.2 Managing secrets in a Docker container

When we created a Docker image for the security token service under the section 10.1, we missed something very important! We embedded all the keys and the credentials to access the keys to the image itself. Now when we push this to a Docker registry, anyone having access to the image can figure out all our secrets, the end of the world! Let's see how it is possible. We have already published the insecure security token service Docker image we created under 10.1 section to the Docker Hub as `prabath/insecure-sts-ch10:v1` and its available to the public. Anyone can execute the following `docker run` command, which will fetch the insecure security token service image and run it locally. If you are new to Docker, appendix A teaches you how to publish a Docker image to Docker Hub.

```
\> docker run -d prabath/insecure-sts-ch10:v1
34839d0c9e3b32b5f4fa6a8b4f7f52c134ed3e198ad739b722ca556903e74fbc
```

Once the container got started, we can use the following command to connect to the container with the container id (use the full container id from the output of the previous `docker run` command), and access the running shell:

```
\> docker exec -it 34839d0c9e3b32b5f4fa6a8b4f7f52c134.. sh
#
```

Now we are on the container's shell and have direct access to its file system. Let's first list all the files under the root of the container file system:

```
# ls
bin                               proc
com.manning.mss.ch10.sample01-1.0.0.jar  root
dev                               run
etc                               sbin
home                             srv
keystore.jks                     sys
lib                                tmp
```

| | |
|-------|-----|
| media | usr |
| mnt | var |
| opt | |

Now we can unzip the `com.manning.mss.ch10.sample01-1.0.0.jar` and find all the secrets in the `application.properties` file:

```
# jar -xvf com.manning.mss.ch10.sample01-1.0.0.jar
# vi BOOT-INF/classes/application.properties
```

The above command will display the content of the `application.properties` file (as shown below), which includes the credentials to access the private key, which is used by the security token service to sign JWTs it issues.

```
server.port: 8443
server.ssl.key-store: /opt/keystore.jks      #A
server.ssl.key-store-password: springboot
server.ssl.keyAlias: spring
spring.security.oauth.jwt: true
spring.security.oauth.jwt.keystore.password: springboot
spring.security.oauth.jwt.keystore.alias: jwtkey
spring.security.oauth.jwt.keystore.name: /opt/jwt.jks      #B
```

#A Keeps the private and public key of the service, which is used in Transport Layer Service (TLS) communication.

#B Keeps the private key, which is used by the STS to sign the JWTs it issues.

10.2.1 Externalizing secrets from Docker images

In this section let's see how to externalize the configuration files from the Docker image we created for security token service. We need to externalize both the keystore files, and the `application.properties` file, which keeps all the secrets. Let's create a directory called `config` under `chapter10/sample01` and move (not just copy – move) the `application.properties` file from `chapter10/sample01/src/main/resources/` to there (`chapter10/sample01/config`). Then, lets run the following two commands (listing 10.2) from `chapter10/sample01` directory to build a new jar file with no `application.properties` file and create a Docker image:

Listing 10.2. The commands to build the STS and create a Docker image

```
\> mvn clean install
[INFO] BUILD SUCCESS

\> docker build -t com.manning.mss.ch10.sample01:v2 -f Dockerfile-2 .
Step 1/4 : FROM openjdk:8-jdk-alpine
--> 792ff45a2a17
Step 2/4 : ADD target/com.manning.mss.ch10.sample01-1.0.0.jar com.manning.mss.ch10.sample01-
1.0.0.jar
--> 2be952989323
Step 3/4 : ENV SPRING_CONFIG_LOCATION=/application.properties
--> Running in 9b62fdebd566
Removing intermediate container 9b62fdebd566
--> 97077304dbdb
Step 4/4 : ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
--> Running in 215919f70683
```

```
Removing intermediate container 215919f70683
--> e7090e36543b
Successfully built e7090e36543b
Successfully tagged com.manning.mss.ch10.sample01:latest
```

You may notice a difference in this command from the command we ran under in listing 10.1 to create a Docker image before. Here (listing 10.2) we pass an extra argument called `-f` with the value `Dockerfile-2`. This is how we can instruct Docker to use our own custom file as the manifest to create a Docker image instead of looking for a file with the name `Dockerfile`. Let's have a look at the content of the `Dockerfile-2` (listing 10.3).

Listing 10.3. The content of the Dockerfile -2

```
FROM openjdk:8-jdk-alpine
ADD target/com.manning.mss.ch10.sample01-1.0.0.jar com.manning.mss.ch10.sample01-1.0.0.jar
ENV SPRING_CONFIG_LOCATION=/opt/application.properties
ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
```

The 1st line of the above `Dockerfile-2`, instructs Docker to fetch the Docker image called, `openjdk:8-jdk-alpine` from Docker registry, and in this case from the public Docker Hub. This is the base image of the Docker image we are about to create. The second line instructs Docker to copy the file `com.manning.mss.ch10.sample01-1.0.0.jar` from the target directory of the host file system to the root of the container file system. The 3rd line instructs Docker to create an environment variable called `SPRING_CONFIG_LOCATION` and point it to `/opt/application.properties` file. The process running inside the container reads this environment variable to find the location of the `application.properties` file and now it looks for the file under the `/opt` directory of the container file system. Finally the 4th line instructs Docker the entry point to the container – or what process to run when we start the container. Unlike the image we created in section 10.1, we do not add any keystore to the image and also there is no `application.properties` file inside the image. When we spin up a container from this image, we need to specify from where in the host file system, the container has to load those two files.

Let's use the following command (from `chapter10/sample01` directory) to spin up a container from the Docker image we just created. If you have carefully looked into the command we used to build the Docker image, we tagged it this time with `v2` – so we need to use the image here with the `v2` tag:

```
\> docker run -p 8443:8443 --mount
    type=bind,source="$(pwd)/keystores/keystore.jks,target=/opt/keystore.jks" --mount
    type=bind,source="$(pwd)/keystores/jwt.jks,target=/opt/jwt.jks" --mount
    type=bind,source="$(pwd)/config/application.properties,target=/opt/application.properties"
        com.manning.mss.ch10.sample01:v2
```

This looks different from the `docker run` command we executed under section 10.1. Here we pass two extra `--mount` arguments. The Docker image we used under section 10.1 to run the container had both `keystore.jks` and `application.properties` file built in. Now, since we do not have those files inside the image, each time we execute `docker run`, we need to tell Docker

how to load those two files from the host file system. That is what the --mount argument does. The first --mount argument binds the keystore/keystore.jks from the host file system to the /opt/keystore.jks in the container file system, the second --mount argument binds the keystore/jwt.jks from the host file system to the /opt/jwt.jks in the container file system, and the third --mount argument binds the /config/application.properties file from the host file system to the /opt/application.properties file in the container file system.

Once we start the container successfully, we see the following logs printed on the terminal:

```
INFO 30901 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
  8443 (https)
INFO 30901 --- [main] c.m.m.ch10.sample01.TokenService          : Started TokenService in
  4.729 seconds (JVM running for 7.082)
```

Now let's test the security token service, with the following cURL command. This is exactly the same cURL command we used under the section 10.1.2.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
  application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=password&username=peter&password=peter123&scope=foo"
  https://localhost:8443/oauth/token
```

10.2.2 Passing secrets as environment variables

Once we externalize the configuration files and keystores from the Docker image, no one will be able to find any secrets in the Docker image. But still we have secrets hardcoded into a configuration file, which we keep in the host file system. Anyone who has access to the host file system will be able to find them. In this section let's see how to remove secrets from the configuration file (application.properties) and pass them to the container as arguments at the time we run it.

Let's copy the content from chapter10/sample01/application.properties file and replace the content in chapter10/sample01/config/application.properties file with it. Listing 10.4 shows the updated content of the chapter10/sample01/config/application.properties file:

Listing 10.4. The content of the sample01/config/application.properties file

```
server.port: 8443
server.ssl.key-store: /opt/keystore.jks
server.ssl.key-store-password: ${KEYSTORE_SECRET}
server.ssl.keyAlias: spring
spring.security.oauth.jwt: true
spring.security.oauth.jwt.keystore.password: ${JWT_KEYSTORE_SECRET}
spring.security.oauth.jwt.keystore.alias: jwtkey
spring.security.oauth.jwt.keystore.name: /opt/jwt.jks
```

Here we have removed all the secrets from application.properties file and replaced them with a placeholder: \${keystore.secret}. Since our change is only in a file we already externalized from the Docker image, we do not need to build a new image. Lets spin up a

container with the following command (run from chapter10/sample01 directory) – with the updated `application.properties` file.

```
\> docker run -p 8443:8443 --mount
    type=bind,source="$(pwd)/keystores/keystore.jks,target=/opt/keystore.jks" --mount
    type=bind,source="$(pwd)/keystores/jwt.jks,target=/opt/jwt.jks" --mount
    type=bind,source="$(pwd)/config/application.properties,target=/opt/application.properties" -e KEYSTORE_SECRET=springboot -e JWT_KEYSTORE_SECRET=springboot
com.manning.mss.ch10.sample01:v2
```

Here we pass the value of the placeholder we kept in the `application.properties` file as an argument to the `docker run` command under the name `-e`. Once we start the container successfully, we see the following logs printed on the terminal:

```
INFO 30901 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
8443 (https)
INFO 30901 --- [main] c.m.m.ch10.sample01.TokenService      : Started TokenService in
4.729 seconds (JVM running for 7.082)
```

Now let's test the security token service, with the following cURL command. This is exactly the same cURL command we used under the section 10.2.1.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
application/x-www-form-urlencoded; charset=UTF-8" -k -d
"grant_type=password&username=peter&password=peter123&scope=foo"
https://localhost:8443/oauth/token
```

10.2.3 Managing secrets in a Docker production deployment

Both the approaches we discussed in sections 10.2.1 and 10.2.2 are foundational to manage secrets in a production, containerized deployment. But none of them provides a clean solution. In both the cases we keep credentials in cleartext. As we discussed at the very beginning of the chapter, when we deploy Docker in a production setup, we do that with some kind of a container orchestration framework. Kubernetes is the most popular container orchestration framework while Docker Swarm is the container orchestration framework built into Docker. Both these frameworks provide better solutions to manage secrets in a containerized environment. In chapter 11 we discuss in detail how to manage secrets with Kubernetes. If you are new to Kubernetes, appendix B provides a comprehensive overview.

10.3 Using Docker Content Trust (DCT) to sign and verify Docker images

In this section we discuss how to use Docker Content Trust (DCT) to sign and verify Docker images. DCT uses Notary for image signing and verification. Notary (<https://github.com/theupdateframework/notary>) is an open source project that does not have a direct dependency on Docker. You can use Notary to sign and verify any content, not necessarily only the Docker images. As a best practice, when you publish a Docker image to a Docker registry (say for example Docker Hub), you need to sign it, so anyone who pulls it can

verify its integrity before use. Notary is an opinionated¹ implementation of The Update Framework (TUF), which we discuss in section 10.3.1, which makes better security easy.

10.3.1 The Update Framework (TUF)

In 2009 a set of security researchers implemented TUF, based on some security flaws identified in Linux package managers. It aimed to help developers maintain the security of a software update system, even against attackers that compromise the repository or signing keys². The first reference implementation of TUF specification is called Thandy, which is an application updater for the popular software Tor (<https://www.torproject.org/>). If you are interested in learning TUF in detail, we recommend having a look at the TUF specification available at <https://github.com/theupdateframework/specification>.

10.3.2 Docker Content Trust

Docker Content Trust (DCT) integrates Notary into Docker in an opinionated way from Docker 1.8 release onward. In doing that DCT does not support all the functions Notary does, but only a subset. If you'd like to do some advanced functions related to key management, you would still need to use Notary – and there is a Notary command-line tool, which comes with the Docker distribution. Once the DCT is setup, it lets developers pull, push, build, create and run Docker commands in the same way they did before, but behind the scene DCT makes sure all the images, which are published to the Docker registry are signed and also only the signed images are pulled from a Docker registry.

When you publish an image to the Docker registry it will be signed by the publisher's (or your) private key and when you interact with a Docker image (via pull, run, etc) for the first time, you establish the trust with the publisher of that image and signature of the image will be verified against the corresponding public key. If you have used SSH, this model of trust bootstrap is similar to that. When you SSH a server for the first time, you are asked whether to trust the server or not and for subsequent interactions, the SSH client remembers your decision.

10.3.3 Generating keys

Let's use the following command to generate keys for signing with DCT. The `docker trust key generate` command generates a public/private key pair, with a key ID and stores the corresponding public key in the file system, under the same directory where you run the command. The key ID is generated by the system and is mapped to the given name of the signer (in this example prabath is the name of the signer):

```
\$ docker trust key generate prabath
Generating key for prabath...
```

¹ An opinionated implementation guides you through to do something in a well-defined way with a concrete set of tool set or in other words, it gives you less options, rather dictates this is how you do it.

² The Update Framework, <https://github.com/theupdateframework.github.io/>

```
Enter passphrase for new prabath key with ID 1a60acb:XXXXX
Repeat passphrase for new prabath key with ID 1a60acb: XXXXX
Successfully generated and loaded private key. Corresponding public key available:
/Users/prabathsiriwardana/dct/prabath.pub
```

The key generated by the above command is called a delegation key. This is the key used to sign all images we publish to a Docker registry. A signer owns the delegation key. Next, we need to associate this signer's public key with a Docker repository (don't get it confused with Docker registry, if you new to Docker please refer appendix A for the definition). Run the following command, from the same location, which you ran the previous one, to associate the public key of the signer with the `prabath/insecure-sts-ch10` repository (we have already created this repository in Docker Hub, with the image we built in section 10.1):

```
\> docker trust signer add --key prabath.pub prabath/insecure-sts-ch10

Adding signer "prabath" to prabath/insecure-sts-ch10...
Initializing signed repository for prabath/insecure-sts-ch10...
You are about to create a new root signing key passphrase. This passphrase will be used to protect the most sensitive key in your signing system. Please choose a long, complex passphrase and be careful to keep the password and the key file itself secure and backed up. It is highly recommended that you use a password manager to generate the passphrase and keep it safe. There will be no way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 494b9b7: XXXXX
Repeat passphrase for new root key with ID 494b9b7: XXXXX
Enter passphrase for new repository key with ID 44f0da3: XXXXX
Repeat passphrase for new repository key with ID 44f0da3: XXXXX
Successfully initialized "prabath/insecure-sts-ch10"
Successfully added signer: prabath to prabath/insecure-sts-ch10
```

When we run the above command for the first time, it generates two more key pairs: the root key pair and the target key pair. The `--key` argument takes the public key (`prabath.pub`) of the signer as the value and then the name of the signer (`prabath`). Finally at the end of the command you can specify one or more repositories delimited by a space. DCT generates a target key pair for each repository. Since we specify only one repository in the above command, it only generates one target key pair. The root key signs each of these target keys. Target keys are also known as repository keys. All the generated private keys corresponding to the above root, target and signer keys are by default available at `~/.docker/trust/private` directory. Following shows the scrambled private keys:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
role: root
=====
MIHuMEkGCSqGSTib3DQEFDTA8MBsGCSqGSTib3DQEFDAA0BAgwNkfrd40JDQICCAw
==

-----END ENCRYPTED PRIVATE KEY-----

-----BEGIN ENCRYPTED PRIVATE KEY-----
gun: docker.io/prabath/manning-sts
role: targets
=====
MIHuMEkGCSqGSTib3DQEFDTA8MBsGCSqGSTib3DQEFDAA0BAhs5CaEbLT65gICCAw
==
```

```
-----END ENCRYPTED PRIVATE KEY-----

-----BEGIN ENCRYPTED PRIVATE KEY-----
role: prabath

MIHuMEkGCSqSIB3DQEFDTA8MBsGCSqSIB3DQEFDAA0BAiX8J+5px9aogICCAAw
==

-----END ENCRYPTED PRIVATE KEY-----
```

10.3.4 Signing with Docker Content Trust

Let's use the following command to sign the `prabath/insecure-sts-ch10` Docker repository with the delegation key, which we generated in the previous section under the name `prabath`. This is in fact the signer's key:

```
\> docker trust sign prabath/insecure-sts-ch10:v1
Signing and pushing trust data for local image prabath/insecure-sts-ch10:v1, may overwrite
remote trust data
The push refers to repository [docker.io/prabath/insecure-sts-ch10]
be39ecbbf21c: Layer already exists
4c6899b75fdb: Layer already exists
744b4cd8cf79: Layer already exists
503e53e365f3: Layer already exists
latest: digest: sha256:a3186dad0b017be1fef8ead32eedf8db8b99a69af25db97955d74a0941a5fb502 size:
1159
Signing and pushing trust metadata
Enter passphrase for prabath key with ID 706043c: XXXXX
Successfully signed docker.io/prabath/insecure-sts-ch10:v1
```

Now we can use the following command to publish the signed Docker image to Docker Hub:

```
\> docker push prabath/insecure-sts-ch10:v1
The push refers to repository [docker.io/prabath/insecure-sts-ch10]
be39ecbbf21c: Layer already exists
4c6899b75fdb: Layer already exists
744b4cd8cf79: Layer already exists
503e53e365f3: Layer already exists
latest: digest: sha256:a3186dad0b017be1fef8ead32eedf8db8b99a69af25db97955d74a0941a5fb502 size:
1159
Signing and pushing trust metadata
Enter passphrase for prabath key with ID 706043c:
Passphrase incorrect. Please retry.
Enter passphrase for prabath key with ID 706043c:
Successfully signed docker.io/prabath/insecure-sts-ch10:v1
```

Once we publish the signed image to the Docker Hub, we can use the following command to inspect the trust data associated with it:

```
\> docker trust inspect --pretty prabath/insecure-sts-ch10:v1

Signatures for prabath/manning-sts
SIGNER          DIGEST          SIGNERS
latest          a3186dad0b017be1fef8ead32eedf8...  prabath

List of signers and their keys for prabath/insecure-sts-ch10:v1

SIGNER          KEYS
prabath         706043cc4ae3
```

```
Administrative keys for prabath/insecure-sts-ch10:v1

Repository Key
44f0da3f488ff4d4870b6a635be2af60bcef78ac15ccb88d91223c9a5c3d31ef
Root Key
5824a2be3b4ffe4703dfa2032255d3cbf434aa8d1839a2e4e205d92628fb247
```

10.3.5 Signature verification with Docker Content Trust

Out of the box content trust is disabled at the Docker client side. To enable it we need to set the `DOCKER_CONTENT_TRUST` environment variable to 1, as shown in the following command:

```
\> export DOCKER_CONTENT_TRUST=1
```

Once content trust is enabled, the Docker client makes sure all the push, build, create, pull and run Docker commands are executed only against signed images. The following command shows, what happens if we try to run an unsigned Docker image.

```
\> docker run prabath/insecure-sts-ch10:v2
docker: Error: remote trust data does not exist for docker.io/prabath/insecure-sts-ch10:v2:
    notary.docker.io does not have trust data for docker.io/prabath/insecure-sts-ch10:v2.
```

To disable content trust, we can override the value of `DOCKER_CONTENT_TRUST` environment variable to empty as shown in the following command:

```
\> export DOCKER_CONTENT_TRUST=
```

10.3.6 Type of keys used in Docker Content Trust

DCT uses five types of keys: the root key, the target key, the delegation key, the timestamp key and the snapshot key. So far we only know about the root key, the target and the delegation key. The figure 10.2 shows the hierarchical relationship between different types keys.

The root key, which is also known as the offline key is the most important key in DCT. It has a long expiration time and must be protected with highest security. It is recommended to keep it offline (that is how the name offline key was derived), possibly in a USB or some kind of an offline device. A developer or an organization owns the root key and uses it to sign other keys in DCT. When you sign a Docker image with signer's key, a set of trust data gets associated with that image, which you can find in your local file system under `~/.docker/trust/tuf/docker.io/[repository_name]/metadata` directory. For example, the metadata of `prabath/insecure-sts-ch10` is under the directory, `~/.docker/trust/tuf/docker.io/prabath/insecure-sts-ch10/metadata`. Following shows the list of files available under the metadata directory:

```
\> cd ~/.docker/trust/tuf/docker.io/prabath/insecure-sts-ch10/metadata
\> ls
root.json snapshot.json targets targets.json          timestamp.json
\> ls targets
prabath.json     releases.json
```

The root key signs the `root.json` file, which lists out all the valid public keys corresponding to `prabath/insecure-sts-ch10` repository.

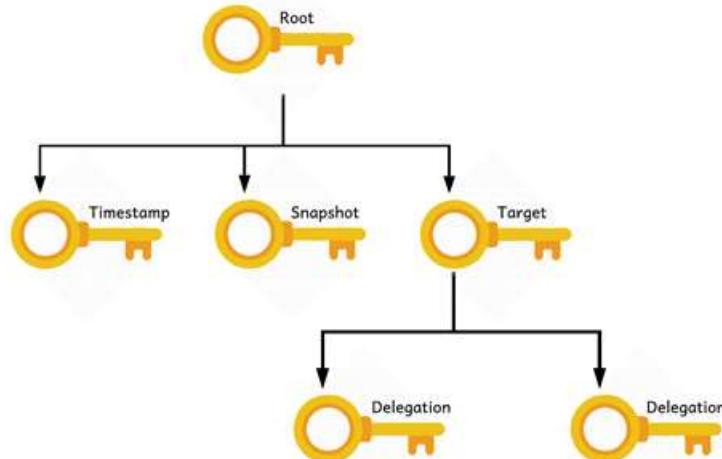


Figure 10.2 Docker Content Trust uses a key hierarchy to sign and verify Docker images

DCT generates a target key per each Docker repository and the root key signs each target key. Once we generate the root key, we only need it again only when we generate target keys. A given repository has one target key – but multiple delegation keys. DCT uses these delegation keys to sign and push images to repositories. The target key signs the `target.json` file (under the `metadata` directory), which lists out all the delegation keys valid for the corresponding repository.

The snapshot key, which is generated by DCT, signs the `snapshot.json` file. The `snapshot.json` file lists out all the valid trust metadata files (except `timestamp.json`) along with the hash of each one. The timestamp key signs the `timestamp.json` file, which carries the hash of the currently valid `snapshot.json` file. The timestamp key has a very short expiration and when each time DCT generates a new key, it re-signs to `timestamp.json` file.

10.4 Running the Order Processing microservice on Docker

In this section, you first build and then deploy the Order Processing microservice on Docker. The Order Processing microservice we use here is the same one, which we used in section 7.7. It is a secured microservice; to access it, we need a valid JWT from the security token service (STS), which we discussed in section 10.1. You can find the complete source code related to the Order Processing microservice in the `chapter10/sample02` directory.

First, build the project with the following Maven command from the `chapter10/sample02` directory. If everything goes well, you should see the `BUILD SUCCESS` message at the end:

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

Now, let's run the following command from chapter10/sample02 directory to build a Docker image for the Order Processing microservice. This command uses the Dockerfile manifest, which is inside the chapter10/sample02 directory:

```
\> docker build -t com.manning.mss.ch10.sample02:v1 .
```

Before we proceed further let's revisit our use case. As illustrated in figure 10.3, here we try to invoke the Order Processing microservice from a token issued by the security token service. The client application first has to get a token from the security token service and then pass it to the Order Processing service. Now the Order Processing talks to the security token service to get its public key, which is corresponding to the private key used by the security token service to sign the token it issued. This is the only communication happens between the Order Processing service and the security token service. If you check the application.properties file under chapter10/sample02/config directory, you will find a property called, security.oauth2.resource.jwt.keyUri, which points to the security token service.

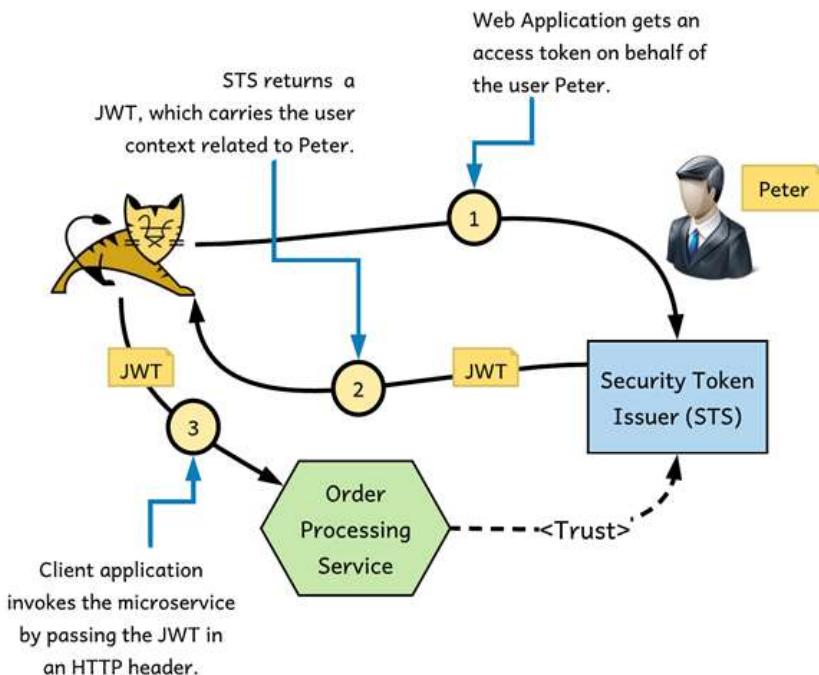


Figure 10.3 STS issues a JWT access token to the web application and the web application uses it to access the microservice, on behalf of the user, Peter.

To enable direct communication between the containers running Order Processing service and the security token service, we need to create user-defined network in the following way. When two Docker containers are in the same user-defined network, they can talk to each other just using the container name. Following command creates a user-defined network called, manning-network. If you are new to Docker, appendix A teaches you more networking options available in Docker:

```
\> docker network create manning-network
06d1307dc12d01f890d74cb76b5e5a16ba75c2e8490c718a67f7d6a02c802e91
```

Now lets spin up the security token service, with the following command (from chapter10/sample01) and attach it to the manning-network we just created:

```
\> docker run --name sts --net manning-network -p 8443:8443 --mount
  type=bind,source="$(pwd)/keystores/keystore.jks,target=/opt/keystore.jks" --mount
  type=bind,source="$(pwd)/keystores/jwt.jks,target=/opt/jwt.jks" --mount
  type=bind,source="$(pwd)/config/application.properties,target=/opt/application.properties"
  -e KEYSTORE_SECRET=springboot -e JWT_KEYSTORE_SECRET=springboot
  com.manning.mss.ch10.sample01:v2
```

Here we use the `--net` argument to specify the name of network and `--name` argument specify the name of the container. Now this container is accessible by any container in the same network by the container name. Also the above command uses the STS image we published to Docker Hub in section 10.2. Make sure that your `--mount` arguments in the above command point to the correct file locations. If you run it from chapter10/sample01, it should just work.

Next let's spin up the Order Processing microservice, from the image we created at the beginning of this section. Execute the following command from chapter10/sample02 directory:

```
\> docker run --net manning-network -p 9443:9443 --mount
  type=bind,source="$(pwd)/keystores/keystore.jks,target=/opt/keystore.jks" --mount
  type=bind,source="$(pwd)/keystores/trust-store.jks,target=/opt/trust-store.jks" -e
  KEYSTORE_SECRET=springboot -e TRUSTSTORE_SECRET=springboot --mount
  type=bind,source="$(pwd)/config/application.properties,target=/opt/application.properties"
  com.manning.mss.ch10.sample02:v1
```

Here we are passing `keystore.jks`, `trust-store.jks` and `application.properties` files as `--mount` arguments. If you look at the `application.properties` file under chapter10/sample02/config directory, you will find a property called, `security.oauth2.resource.jwt.keyUri`, which points to the endpoint `https://sts:8443/oauth/token_key`, having the name of the security token service's container (`sts`) as the host name.

To invoke the Order Processing microservice with proper security, you need to get a JWT from the security token service, using the following curl command. For clarity, we removed the long JWT in the response and replaced it with the value `jwt_access_token`:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
  application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=password&username=peter&password=peter123&scope=foo"
  https://localhost:8443/oauth/token
```

```
{
"access_token":"jwt_access_token",
"token_type":"bearer",
"refresh_token":"",
"expires_in":1533280024,
"scope":"foo"
}
```

Now try to invoke the Order Processing microservice with the JWT you got from the above curl command. Set the same JWT we got, in the HTTP Authorization Bearer header using the following curl command, and invoke the Order Processing microservice. Because the JWT is little lengthy, you can use a small trick when using the curl command. First, export the JWT to an environmental variable (`TOKEN`). Then use that environmental variable in your request to the Order Processing microservice.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" https://localhost:9443/orders/11

{"customer_id":"101021","order_id":"11","payment_method":{"card_type":"VISA","expiration":"01/22","name":"John Doe","billing_address":"201, 1st Street, San Jose, CA"}, "items":[{"code":"101","qty":1}, {"code":"103","qty":5}], "shipping_address":"201, 1st Street, San Jose, CA"}
```

10.5 Running containers with limited privileges

In any operating system, there is a super user or an administrator, who can basically do anything. This user is called root, in most of the Linux distributions. Traditionally in the Linux kernel, there are two types of processes: privileged processes and unprivileged processes. Any privileged process runs with the special user ID, 0 – which belongs to the root user. Any process carrying non-zero user ID is an unprivileged process. A privileged process bypasses all the kernel level permission checks, while all the unprivileged processes are subjected to a permission check. This approach gives too much of power to the root user – in any case, too much of power is too dangerous.

All Docker containers, by default run as the root user. Does that mean anyone having access to the container can do anything to the host file system, from a container? In appendix A, under the section A.13.4, we discuss how Docker brings process isolation with six namespaces. A namespace in Linux partitions kernel resources, so that each running process will have its own independent view of those resources. The mount namespace (one of the six namespaces) helps isolating one container's view of the file system from other containers', as well as the host file system. Each container will see its own /usr, /var, /home, /opt, /dev directories. Any change you do as the root user within a container will still remain inside the container file system. But, when you use a volume (appendix A, section A.12), which maps a location in the container file system to the host file system, the root user can be destructive. Also, an attacker having access to a container running as root can use root privileges to install some tools within the container, and use those tools to find any vulnerability in other services in the network.

In the following sections, we explore what options are available to run a container as a non-privileged process.

10.5.1 Running a container with a non-root user

There are two approaches to run a container with a non-root user. One way is to use the `--user` (or `-u`) flag in the `docker run` command and the other way is to define the user who you want to run the container in the Dockerfile itself.

Let's see how the first approach works. In the following command, we start a Docker container from the `prabath/insecure-sts-ch10:v1` image, which we already published to Docker Hub.

```
\> docker run --name insecure-sts prabath/insecure-sts-ch10:v1
```

Let the above container keep running, and use the following command from a different terminal to connect to the file system of the running container. The `insecure-sts` is the name of the container we started in the previous command.

```
\> docker exec -it insecure-sts sh  
#
```

Now you are connected to the container file system. You can try out any commands available in alpine linux there. The command `id`, tells you the user ID (uid) and the group ID (gid) of the user who runs the container.

```
# id  
uid=0(root) gid=0(root)
```

Let's remove the `insecure-sts` with the following command, run from a different terminal.

```
\> docker rm insecure-sts
```

Following command runs the `insecure-sts` from the `prabath/insecure-sts-ch10:v1` image, with the `--user` flag. This flag instructs Docker to run the container with the user having the user id, 1000 and group id 800.

```
\> docker run --name insecure-sts --user 1000:800 prabath/insecure-sts-ch10:v1
```

Let the above container keep running, and use the following command from a different terminal to connect to the file system of the running container and find out the user who runs the container.

```
\> docker exec -it insecure-sts sh  
# id  
uid=1000 gid=800
```

The second approach to run a container as a non-root user is to define the user who we want to run the container in the Dockerfile itself. This is a good approach if you are the developer who builds Docker images, but won't help if you are just the user. The first approach helps in such case.

Let's have a look at the Dockerfile (listing 10.5), we used in section 10.1. You can find the source code related to this sample inside chapter10/sample01 directory.

Listing 10.5. The content of the Dockerfile

```
FROM openjdk:8-jdk-alpine
ADD target/ com.manning.mss.ch10.sample01-1.0.0.jar com.manning.mss.ch10.sample01-1.0.0.jar
ENV SPRING_CONFIG_LOCATION=/application.properties
ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
```

We have no instruction to define a user to run this container in the Dockerfile. In such case, Docker looks for the base image, which is `openjdk:8-jdk-alpine`. You can use the following `docker inspect` command to find out the details. It produces a lengthy output and if you look for the `User` element under the `ContainerConfig` element, you can find out who the user is.

```
\$ docker inspect openjdk:8-jdk-alpine
[{"ContainerConfig": {"User": ""}}]
```

According to the above output, even the base image (`openjdk:8-jdk-alpine`) does not instruct Docker to run the corresponding container as non-root user. In such case, by default, Docker uses root user to run the container. To fix that, we update our Dockerfile (listing 10.6) with the `USER` instruction, which asks Docker to run the corresponding container as a user with the user id, 1000.

Listing 10.6. The updated content of the Dockerfile with USER instruction

```
FROM openjdk:8-jdk-alpine
ADD target/ com.manning.mss.ch10.sample01-1.0.0.jar com.manning.mss.ch10.sample01-1.0.0.jar
ENV SPRING_CONFIG_LOCATION=/application.properties
USER 1000
ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
```

10.5.2 Dropping capabilities from the root user

Linux kernel 2.2 introduced a new feature called, capabilities. It divides all the privileged operations a root user can perform into a set of capabilities. For example, `CAP_CHOWN` capability lets a user to execute the `chown()` operation, which can be used to change the user ID (UID) and/or group ID (GID) of a file. All these capabilities can be independently enabled or disabled on the root user. This approach let's you start a Docker container as the root user, but with a limited set of privileges. Let's use the Docker image we created in section 10.1 to experiment this approach. Following command starts a Docker container from the `prabath/insecure-sts-ch10:v1` image, which we already published to Docker Hub.

```
\> docker run --name insecure-sts prabath/insecure-sts-ch10:v1
```

Let the above container keep running, and use the following command (as in section 10.5.1) from a different terminal to connect to the file system of the running container, and find out the user ID (uid) and the group ID (gid) of the user who runs the container.

```
\> docker exec -it insecure-sts sh
# id
uid=0(root) gid=0(root)
```

To find out what capabilities the root user has on the system, we need to run a tool called, getpcaps, which comes as part of the libcap package. Since the default distribution of alpine linux does not have this tool, we use alpine package manager (apk) to install libcap, with the following command. Since we are still inside the container file system, this installation has no impact on the host file system.

```
# apk add libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.9/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.9/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.26-r0)
Executing busybox-1.29.3-r10.trigger
OK: 103 MiB in 55 packages
```

Once the installation is completed successfully, we use the following command to find out the capabilities associated with root user.

```
# getpcaps root
Capabilities for `root': =
    cap_chown, cap_dac_override, cap_fowner, cap_fsetid, cap_kill, cap_setgid, cap_setuid, cap_setpcap, cap_net_bind_service, cap_net_raw, cap_sys_chroot, cap_mknod, cap_audit_write, cap_setfcap+eip
```

Let's remove the `insecure-sts` with the following command, run from a different terminal.

```
\> docker rm insecure-sts
```

Following command runs the `insecure-sts` container from the `prabath/insecure-sts-ch10:v1` image, with the `--cap-drop` flag. This flag instructs Docker to drop the chown capability from the root user, who runs the container. The Linux kernel prefixes all capability constants with "cap_", for example, `cap_chown`, `cap_kill`, `cap_setuid`, and so on. Docker capability constants are not prefixed with "cap_" but otherwise match the kernel's constants, for example just `chown`, instead of `cap_chown`.

```
\> docker run --name insecure-sts --cap-drop chown prabath/insecure-sts-ch10:v1
```

Let the above container keep running, and use the following command from a different terminal to connect to the file system of the running container.

```
\> docker exec -it insecure-sts sh
```

Since we have started a new container, and since the container file system is immutable, we need to install libcap again, using the following command.

```
# apk add libcap
```

If you check the capabilities of the root user again, you see that the `cap_chown` capability is not there.

```
# getpcaps root
Capabilities for `root': =
    cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_n
et_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap+eip
```

One main benefit of capabilities is that you need not to know the user who runs the container. The capabilities you define in the `docker run` command are applicable to any user who runs the container.

Just like how we dropped some capabilities in the `docker run` command, we also can add. The following command drops all the capabilities and adds only one capability.

```
\> docker run --name insecure-sts --cap-drop ALL --cap-add audit_write prabath/insecure-sts-
ch10:v1
```

10.6 Running Docker Bench for security

Docker Bench for Security is a script, which checks a Docker deployment for common, well-known best practices as defined by Center for Internet Security (CIS) in Docker Community Edition Benchmark document (<https://downloads.cisecurity.org>). It is maintained as an open source project under the git repository: <https://github.com/docker/docker-bench-security>. This script can be executed either by itself or as a Docker container. The command below follows the 2nd approach, where we run Docker Bench for Security, with the Docker image `docker/docker-bench-security`. It checks the Docker host configuration, Docker daemon configuration, all the container images available in the host machine, and container runtimes. Here we have truncated the output only to show you the important areas at a high-level covered by Docker for Security Bench:

```
\> docker run -it --net host --pid host --cap-add audit_control -v /var/lib:/var/lib -
    /var/run/docker.sock:/var/run/docker.sock -v /etc:/etc --label docker_bench_security
    docker/docker-bench-security

# -----
# Docker Bench for Security v1.3.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best practices around deploying Docker containers in
# production.
# Inspired by the CIS Docker 1.13 Benchmark.
# -----
[INFO] 1 - Host Configuration
[WARN] 1.1 - Create a separate partition for containers
[INFO] 1.2 - Harden the container host
[PASS] 1.3 - Keep Docker up to date

[INFO] 2 - Docker Daemon Configuration
[WARN] 2.1 - Restrict network traffic between containers
```

```
[PASS] 2.2 - Set the logging level
[PASS] 2.3 - Allow Docker to make changes to iptables

[INFO] 3 - Docker Daemon Configuration Files
[INFO] 3.1 - Verify that docker.service file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.2 - Verify that docker.service file permissions are set to 644 or more restrictive
[INFO]      * File not found

[INFO] 4 - Container Images and Build Files
[WARN] 4.1 - Create a user for the container
[WARN]      * Running as root: affectionate_lichterman
[INFO] 4.2 - Use trusted base images for containers

[INFO] 5 - Container Runtime
[WARN] 5.1 - Do not disable AppArmor Profile
[WARN]      * No AppArmorProfile Found: affectionate_lichterman
[WARN] 5.2 - Verify SELinux security options, if applicable

[INFO] 6 - Docker Security Operations
[INFO] 6.1 - Perform regular security audits of your host system and containers
[INFO] 6.2 - Monitor Docker containers usage, performance and metering
```

10.7 Securing access to Docker host

In appendix A, under section A.3 we discuss the high-level architecture of Docker. If you are new to Docker, we recommend you go through it first. Figure 10.4 (copied from appendix A) illustrates the communication between a Docker client and a Docker host. If you want to intercept the communication between the client and host and see what's going on, you can use a tool like socat (see appendix A, section A.15). So far in this chapter, most of the Docker commands we used via Docker client, assumed we run both the Docker client and the daemon in the same machine.

In this section we discuss how to setup a Docker host to accept requests securely from a remote Docker client. In practice this is not the case, when we run Docker with Kubernetes (see appendix B). You do not need direct access to the Docker daemon – but only to the Kubernetes API server. In chapter 11, we discuss securing access to Kubernetes API server. But, still there are many who run Docker without Kubernetes. For example, the CI/CD (continuous integration/continuous delivery) tools used to spin up Docker containers, connecting remotely to a Docker host. In such cases, we need to expose Docker daemon securely to remote clients.

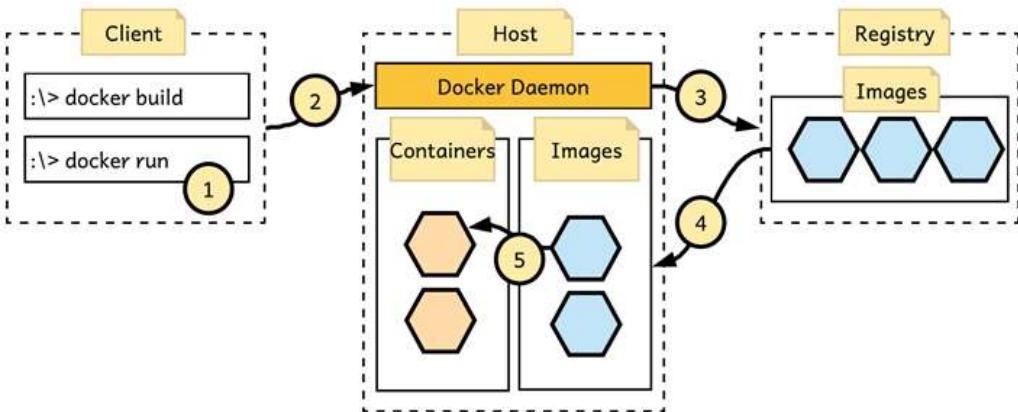


Figure 10.4 High-level Docker component architecture. The Docker client talks to the Docker daemon running on the Docker host over a REST API to perform various operations on Docker images and containers.

10.7.1 Enabling remote access to Docker daemon

Docker daemon supports listening on three types of sockets: Unix, TCP and FD (file descriptor). By enabling TCP socket will let your Docker client talks to the daemon remotely. But, if you run Docker host on Mac, you will find it hard to enable TCP socket on Docker daemon. Here we follow a workaround – that will work across any operating system and also gives you more flexibility to control access to Docker APIs. The figure 10.5 illustrates what we want to build.

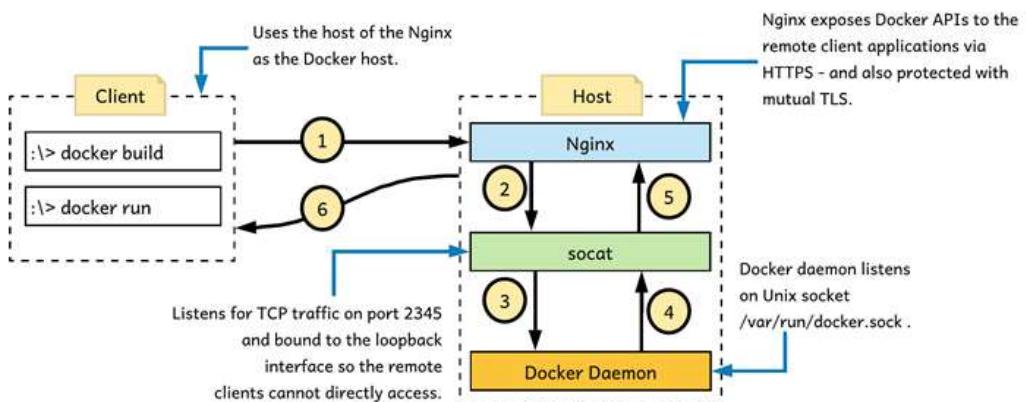


Figure 10.5 Exposing Docker APIs securely to remote clients via Nginx. Socat is used a traffic forwarder between Nginx and Docker daemon.

As per figure 10.5, we are running socat as a traffic-forwarder, on the same machine, which runs Docker daemon. It listens for TCP traffic on port 2345 and forwards to the Unix socket, which Docker daemon listens to. Then we have an Nginx (<http://nginx.org/en/docs/>) instance, which acts as a reverse proxy to socat. All the external Docker clients talk to the Docker daemon via Nginx. In this section we are going to setup Nginx and socat, with Docker compose. If you are new to Docker compose, we recommend you go through the section A.16 of appendix A. Listing 10.7 shows the complete `docker-compose.yaml` file. You can find the same under `chapter10/sample03` directory.

Listing 10.7. The content of the docker-compose.yaml file

```
version: '3'
services:
  nginx:
    image: nginx:alpine
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    ports:
      - "8080:8080"
    depends_on:
      - "socat"
  socat:
    image: alpine/socat
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
      - "2345:2345"
    command: TCP-L:2345,fork,reuseaddr,bind=socat UNIX:/var/run/docker.sock
```

The listing 10.7 defines two services, one for the Nginx and the other for the socat. The Nginx service uses `nginx:alpine` Docker image and socat service uses `alpine/socat` Docker image. For the Nginx image, we have a volume mount, which mounts `nginx.conf` from (`chapter10/sample03`) directory of the host file system to the `/etc/nginx/nginx.conf` file of the container file system. This is the main Nginx configuration file, which forwards all the traffic it gets to socat. Listing 10.8 shows the Nginx configuration.

Listing 10.8. The content of the nginx.conf file

```
events {}
http {
  server {
    listen 8080;
    location / {
      proxy_pass      http://socat:2345/;
      proxy_redirect  off;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Host $server_name;
    }
  }
}
```

For the socat image, we have a volume mount, which mounts `/var/run/docker.sock` from the host file system to the `/var/run/docker.sock` of the container file system. This is the file, which represents the Unix socket Docker daemon listens to, on the host machine. When we do this volume mounting, the container, which runs socat, can write directly to the Unix socket on the host file system - so that the Docker daemon gets the messages. Let's have a look at the following line, which is the last line in listing 10.7.

```
command: TCP-L:2345,fork,reuseaddr,bind=socat UNIX:/var/run/docker.sock
```

The `TCP-L:2375` flag instructs socat to listen on port 2375 for TCP traffic. The `fork` flag enables socat to handle each arriving packet by its own sub process. In other words, when we use `fork`, socat creates a new process for each newly accepted connection. The `bind=127.0.0.1` flag instructs socat to listen only on the loopback interface, so no one outside the host machine can directly talk to socat. The `UNIX:/var/run/docker.sock` is the address of the Unix socket, where the Docker daemon accepts connections. In effect, the above command asks socat to listen for TCP traffic on port 2345, log them and forward them to the Unix socket `/var/run/docker.sock`.

Let's run the following command from `chapter10/sample03` directory to start both the Nginx and socat containers:

```
\> docker-compose up
Pulling socat (alpine/socat:)... 
latest: Pulling from alpine/socat
ff3a5c916c92: Pull complete
abb964a97c4c: Pull complete
Pulling nginx (nginx:alpine)...
alpine: Pulling from library/nginx
e7c96db7181b: Already exists
f0e40e45c95e: Pull complete
Creating sample03_socat_1 ... done
Creating sample03_nginx_1 ... done
Attaching to sample03_socat_1, sample03_nginx_1
```

To make sure everything works fine, you can run the following command from the Docker client machine, with the proper Nginx host name. It should return you back a JSON payload, which carries Docker image details.

```
\> curl http://nginx-host:8080/v1.39/images/json
```

10.7.2 Enabling mTLS at the Nginx server to secure access to Docker APIs

In this section we see how to secure the APIs exposed by the Nginx server with mTLS, so that all the Docker APIs will be secured too. To do that, we need to create a public/private key pair for the Nginx server and also for the Docker client. The Docker client will use its key pair to authenticate to the Nginx server.

GENERATING KEYS AND CERTIFICATES FOR THE NGINX SERVER AND THE DOCKER CLIENT

Here we introduce a single script to perform all the actions to create keys for the certificate authority, Nginx server, and Docker client. The certificate authority signs both the Nginx

certificate and Docker client's certificate. We run OpenSSL in a Docker container to generate keys. OpenSSL is a commercial-grade toolkit and cryptographic library for TLS, available for multiple platforms. You can refer appendix K to find more details on OpenSSL and key generation. To spin up the OpenSSL Docker container, run the following docker run command from the chapter10/sample03/keys directory.

```
\> docker run -it -v $(pwd):/export prabath/openssl
#
```

This docker run command starts OpenSSL in a Docker container, with a volume mount, which maps chapter10/sample03/keys (or the current directory, which is indicated by \$(pwd)) directory from the host file system to the /export directory of the container file system. This volume mount helps you to share part of the host file system with the container file system. When the OpenSSL container generates certificates, those are written to the /export directory of the container file system. Since we have a volume mount, everything inside the /export directory of the container file system is also accessible from the chapter10/sample03/keys directory of the host file system.

When you run the command above for the first time, it may take couple of minutes to execute and ends with a command prompt, where we can execute our script to create all the keys.

```
# sh /export/gen-key.sh
```

Now, if you look at the chapter10/sample03/keys directory in the host file system, you will find the following set of files. If you want to understand, what happens underneath the script, please check appendix K.

- ca_key.pem and ca_cert.pem files under chapter10/sample03/keys/ca directory. ca_key.pem is the private key of the certificate authority, and ca_cert.pem is the public key.
- nginx_key.pem and nginx_cert.pem files under chapter10/sample03/keys/nginx directory. nginx_key.pem is the private key of the Nginx server, and nginx_cert.pem is the public key. The certificate authority signs Nginx keys.
- docker_key.pem and docker_cert.pem files under chapter10/sample03/keys/docker directory. docker_key.pem is the private key of the Docker client, and docker_cert.pem is the public key. The certificate authority signs the Docker keys and Docker uses these keys to authenticate to the Nginx server

PROTECTING NGINX SERVER WITH MTLS

In this section we are going to setup Nginx to work with mutual TLS (mTLS), with the keys we generated in the previous section. If you are running the Nginx container from section 10.7.1, we have to stop it first, by pressing Ctrl+C on the terminal, which runs the container.

Listing 10.9 shows the content from `nginx-secured.conf` file inside `chapter10/sample03` directory. This is the same file in listing 10.8, with some new parameters related to the TLS configuration. The parameter `ssl_certificate` instructs Nginx to look for the server certificate at `/etc/nginx/nginx_cert.pem` location, in the container file system.

Listing 10.9. The content from the nginx-secured.conf file

```
events {}
http {
    server {
        listen          8443 ssl;
        server_name     nginx.ecomm.com;
        ssl_certificate /etc/nginx/nginx_cert.pem;
        ssl_certificate_key /etc/nginx/nginx_key.pem;
        ssl_protocols   TLSv1.2;
        ssl_verify_client on;
        ssl_client_certificate /etc/nginx/ca_cert.pem;
        location / {
            proxy_pass      http://socat:2345/;
            proxy_redirect  off;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Host $server_name;
        }
    }
}
```

Since we keep all the key files in the host file system, in the updated docker-compose configuration (listing 10.10), we have a new set of volume mounts. From the host file system `sample03/keys/nginx/nginx_cert.pem` is mapped to the `/etc/nginx/nginx_cert.pem` location, in the container file system. In the same way we have a volume mount for the private key (`ssl_certificate_key`) of the Nginx server. To enable mutual TLS, we have set the value of `ssl_verify_client` to `on`, in listing 10.8 and `ssl_client_certificate` parameter points to a file, which carries the public keys of all trusted certificate authorities. In other words, we allow any client to access the Docker API, if it brings a certificate issued by a trusted certificate authority.

Now, we need to update docker-compose configuration to use the new `nginx-secured.conf` file. Listing 10.10 shows the updated docker-compose configuration, and also available at `chapter10/sample03/docker-compose-secured.yaml` file.

Listing 10.10. The content from the nginx-secured.conf file

```
services:
  nginx:
    image: nginx:alpine
    volumes:
      - ./nginx-secured.conf:/etc/nginx/nginx.conf
      - ./keys/nginx/nginx_cert.pem:/etc/nginx/nginx_cert.pem
      - ./keys/nginx/nginx_key.pem:/etc/nginx/nginx_key.pem
      - ./keys/ca/ca_cert.pem:/etc/nginx/ca_cert.pem
```

```

ports:
- "8443:8443"
depends_on:
- "socat"
socat:
image: alpine/socat
volumes:
- /var/run/docker.sock:/var/run/docker.sock
ports:
- "2345:2345"
command: TCP-L:2345,fork,reuseaddr,bind=socat UNIX:/var/run/docker.sock

```

Let's run the following command from `chapter10/sample03` directory to start both the secured Nginx and socat containers. This command points the `docker-compose-secured.yaml` file, which carries the new docker-compose configuration.

```
\> docker-compose -f docker-compose-secured.yaml up
```

To make sure everything works fine, you can run the following command from the `chapter10/sample03` directory of the Docker client machine, with the proper Nginx host name. Here we use `-k` option to instruct curl to ignore any HTTPS server certificate validation. Still, this command will fail, because we have now secured all Docker APIs with mutual TLS.

```
\> curl -k https://nginx-host:8443/v1.39/images/json
```

Following command shows, how to use curl with proper client side certificates. Here we use the key pair, which we generated for the Docker client. This should return you back a JSON payload, which carries Docker image details.

```
\> curl --cacert keys/ca/ca_cert.pem --cert keys/nginx/nginx_cert.pem --key
keys/nginx/nginx_key.pem --resolve 'nginx.ecomm.com:8443:127.0.0.1'
https://nginx.ecomm.com:8443/v1.39/images/json
```

The `cacert` argument in the above command points to the public key of the certificate authority, and the `cert` and the `key` parameters point to respectively to the public key and the private key we generated in the previous section for the Docker client. In the API endpoint, the host name we use, must match the common name (CN) of the certificate we use at the Nginx, otherwise the certificate validation will fail. Then again, since we do not have a DNS entry for this host name, we instruct curl to resolve it to the IP address 10.0.0.128 using the `resolve` argument – you probably can use 127.0.0.1 as the IP address if you run curl from the same machine where Docker daemon runs.

CONFIGURING DOCKER CLIENT TO TALK TO THE SECURED DOCKER DAEMON

In this section we are going to configure Docker client to talk to the Docker daemon, via the secured Nginx server. The following command instructs Docker client to use, `nginx.ecomm.com:8443` as the Docker host and the port.

```
\> export DOCKER_HOST=nginx.ecomm.com:8443
```

Since we have not setup nginx.ecomm.com in a DNS server, we need to update the /etc/hosts file of the machine, which runs the Docker client, with a hostname to IP address mapping, as shown below. If you run both the Docker daemon and the client on the same machine, you can use 127.0.0.1 as the IP address of the Docker daemon.

```
10.0.0.128 nginx.ecomm.com
```

Run the following Docker client command from the same terminal, which you exported the DOCKER_HOST environment variable. The `tlsverify` argument instructs Docker client to use TLS to connect to the Docker daemon, and verify the remote certificate. The `tlskey` and `tlscert` arguments point to the private key and the public key of the Docker client respectively. These are the keys that we generated in the previous section. The `tlscacert` argument points to the public key of the certificate authority.

```
\> docker --tlsverify --tlskey keys/docker/docker_key.pem --tlscert
      keys/docker/docker_cert.pem --tlscacert keys/ca/ca_cert.pem images
```

If you want to make the above command looks little simple, we can replace the default keys come with the Docker client. Replace `~/.docker/key.pem` file with `keys/docker/docker_key.pem`, `~/.docker/cert.pem` file with `keys/docker/docker_cert.pem`, and `~/.docker/ca.pem` with `keys/ca/ca_cert.pem`. Now you can run your Docker client command as shown below.

```
\> docker --tlsverify images
```

Ten layers of container security

This RedHat whitepaper (<https://www.redhat.com/en/resources/container-security-openshift-cloud-devops-whitepaper>) talks about ten layers of container security: container host multi-tenancy, container content, container registries, building containers, deploying containers, container orchestration, network isolation, storage, API management, and federated clusters. Though the focus of this whitepaper is mostly on the Red Hat Enterprise Linux and Red Hat OpenShift platform, still it is an excellent read.

10.8 Security beyond containers

In a typical microservices deployment, containers do not act along. Even though there are some who deploy microservices just with containers most of the scalable microservices deployments use containers within a container orchestration framework, such as Kubernetes. Securing a microservices deployment highly depend on the security constructs provided by your container orchestration framework. There are few container orchestration frameworks, but the world is defaulting to Kubernetes. We discuss Kubernetes in detail in appendix B and chapter 11.

Apart from Kubernetes, the service mesh also plays a key role in securing a microservices deployment. A service mesh is a decentralized application-networking infrastructure between microservices that provides, resiliency, security, observability, and routing control. If you are

familiar with software-defined networks (SDN), you can think of a service mesh as an SDN too. There are multiple popular service mesh implementations, but the mostly used one is Istio. In appendix C and chapter 12, we discuss in detail how to secure microservices with Istio.

10.9 Summary

- Docker containers have become the de facto standard to package, distribute, test and deploy microservices.
- Docker uses cgroups and namespaces, which are two basic constructs in the Linux kernel to build an isolated environment to create and run containers.
- As a best practice, no secrets must be embedded into Docker images – and must be externalized. Container orchestration frameworks such as Kubernetes and Docker Swarm provide better ways to manage secrets in a containerized environment.
- You should not think about container security in isolation, in securing microservices. The security of a microservices deployment should be thought of under the context of a container orchestration framework, not just about container security in isolation.
- Docker Container Trust (DCT) is used to sign and verify Docker images. This makes sure that you only run trusted containers in your deployment and also helps developers who rely on Docker images developed by you, to validate them.
- As a best practice you should not run a container as the root user. One approach is to define the user who we want to run the container in the Dockerfile itself or pass it as an argument to the `docker run` command. The other approach is to use capabilities to restrict what a user can do within a container.
- Sometimes you need to expose Docker daemon to remote clients. In such cases you must protect all Docker APIs to make sure only the legitimate users have access.
- Docker Bench for Security checks a Docker deployment for common, well-known best practices. It validates your Docker environment against the well-known best practices as defined by Center for Internet Security (CIS) in Docker Community Edition Benchmark document (<https://downloads.cisecurity.org>).

11

Securing microservices on Kubernetes

This chapter covers

- Securing service-to-service communication of a microservice deployment
- Managing secrets in a deployment
- Creating service accounts and associating those with pods
- Protecting access to the Kubernetes API server with role-based access control (RBAC)

In chapter 10 we discussed how to deploy and secure microservices on Docker containers. In a real production deployment, you don't have only containers; containers are used within a container orchestration framework. As a container is an abstraction over the physical machine, the container orchestration framework is an abstraction over the network. Kubernetes is the most popular container orchestration framework to date.

Understanding the fundamentals of Kubernetes and its security features is essential to any microservices developer. We cover basic constructs of Kubernetes in appendix B, so if you're new to Kubernetes, read appendix B first. Even if you're familiar with Kubernetes, we still recommend you at least skim through appendix B. The rest of the chapter assumes you have the knowledge contained in appendix B.

11.1 Running security token services (STSs) on Kubernetes

In this section we deploy the Docker container that we built in chapter 10 with the security token service (STS) in Kubernetes. This Docker image is already published to the Docker Hub as `prabath/insecure-sts-ch10:v1`. To deploy a container in Kubernetes, first we need to create a

pod. If you read appendix B, you learned that developers or DevOps don't directly work with pods but with deployments. But to create a pod in Kubernetes, we need to create a deployment.

11.1.1 Defining a Kubernetes deployment for the STS in yaml

A deployment is a Kubernetes object that we represent in a yaml file. Let's create the following yaml file (listing 11.1) with the prabath/insecure-sts-ch10:v1 Docker image. The source code related to all the samples in this chapter is available in the GitHub repository at <https://github.com/microservices-security-in-action/samples> in the chapter11 directory. You can also find the same yaml configuration shown in listing 11.1 in the chapter11/sample01/sts.deployment.yaml file.

Listing 11.1. The content of the sts.deployment.yaml file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sts-deployment
  labels:
    app: sts
spec:
  replicas: 1 #A
  selector:
    matchLabels:
      app: sts #B
  template: #C
    metadata:
      labels:
        app: sts
    spec:
      containers:
        - name: sts
          image: prabath/insecure-sts-ch10:v1
          ports:
            - containerPort: 8443
```

#A Instructs Kubernetes to run one replica of the matching pods

#B This deployment will carry a matching pod as per the selector. This is an optional section, which can carry multiple labels.

#C A template describes how each pod in the deployment should look. If you define a selector/matchLabels, then the pod definition must carry a matching label.

11.1.2 Creating the STS deployment in Kubernetes

In this section we create a deployment in Kubernetes for the STS that we defined in the yaml file in section 11.1.1. We assume you've access to a Kubernetes cluster. If not, follow the instructions in appendix B, section B.6.4, to create a Kubernetes cluster with the Google

Kubernetes Engine (GKE).¹ Once you've access to a Kubernetes cluster, go to the chapter11/sample01 directory and run the following command from your local machine to create a deployment for STS:

```
\> kubectl apply -f sts.deployment.yaml  
deployment.apps/sts-deployment created
```

Use the following command to find all the deployments in your Kubernetes cluster (under the current namespace). If everything goes well, you should see one replica of the STS up and running.

```
\> kubectl get deployment sts-deployment  
NAME      READY   UP-TO-DATE   AVAILABLE   AGE  
sts-deployment 1/1     1          1           12s
```

11.1.3 Troubleshooting the deployment

Not everything goes fine all the time. Multiple things can go wrong. In case Kubernetes complaints about the yaml file, it could be due to an extra space or some error when you copy and paste the content from the text on the book. Rather than copying and pasting from the book, always use the corresponding sample file from the GitHub repo. Also, in case you have doubts about your yaml file, you can use an online tool like <http://www.yamllint.com/> to validate it.

Even though the `kubectl apply` command executes successfully, when you run `kubectl get deployments`, it may show that none of your replicas are ready. The following three commands are quite useful in such cases:

- The `kubectl describe` command shows a set of metadata related to the deployment:

```
\> kubectl describe deployment sts-deployment
```

- The `kubectl get events` command shows all the events created in the current Kubernetes namespace. If something goes wrong while creating the deployment, you'll notice a set of errors or warnings.

```
\> kubectl get events
```

- Another useful command in troubleshooting is `kubectl logs`. You can run this command against a given pod. First, though, you can run `kubectl get pods` to find the name of the pod you want to get the logs from, and then use the following command with the pod name (sts-deployment-799fdff46f-hdp5s is the pod name in the following command):

¹All the examples in this book use Google Cloud, which is more straightforward and hassle-free when trying out the examples.

```
\> kubectl logs sts-deployment-799fdff46f-hdp5s --follow
```

Once you identify the issue related to your Kubernetes deployment and if you need help to get that sorted out, either you can reach out any to the Kubernetes community forums (<https://discuss.kubernetes.io>) or use the Kubernetes Stack Overflow channel (<https://stackoverflow.com/questions/tagged/kubernetes>).

11.1.4 Exposing the STS outside the Kubernetes cluster

In this section we create a Kubernetes service that exposes the STS outside the Kubernetes cluster. If you're new to Kubernetes services, remember to check appendix B.9.

Here we use a Kubernetes service of LoadBalancer type. If there are multiple replicas of a given pod, the service of LoadBalancer type acts as a load balancer. Usually it's an external load balancer provided by the Kubernetes hosting environment, but in our case, it's the GKE. Let's have a look at the yaml file to create the service (listing 11.2). The same yaml file is available at chapter11/sample01/sts.service.yaml.

The service listens on port 443 and forwards the traffic to port 8443. If you look at listing 11.1, you'll notice that when we create the deployment, the container that carries the STS microservice is listening on port 8443.

Listing 11.2. The content of the sts.service.yaml file

```
apiVersion: v1
kind: Service
metadata:
  name: sts-service
spec:
  type: LoadBalancer
  selector:
    app: sts
  ports:
    - protocol: TCP
      port: 443
      targetPort: 8443
```

To create the service in the Kubernetes cluster, go to the chapter11/sample01 directory and run the following command from your local machine:

```
\>kubectl apply -f sts.service.yaml
service/sts-service created
```

Use the following command to find all the services in your Kubernetes cluster (under the current namespace):²

```
\>kubectl get services
```

²If you're new to the namespace concept in Kubernetes, check appendix B. All the samples in this chapter use the default namespace.

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-------------|--------------|---------------|-------------|---------------|------|
| kubernetes | ClusterIP | 10.39.240.1 | <none> | 443/TCP | 134m |
| sts-service | LoadBalancer | 10.39.244.238 | <pending> | 443:30993/TCP | 20s |

It takes Kubernetes a few minutes to assign an external IP address for the `sts-service` we just created. If you run the same command after couple of minutes, you'll notice the following output, with an external IP address assigned to the `sts-service`:

| > kubectl get services | | | | | |
|------------------------|--------------|---------------|-------------|---------------|------|
| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
| kubernetes | ClusterIP | 10.39.240.1 | <none> | 443/TCP | 135m |
| sts-service | LoadBalancer | 10.39.244.238 | 34.82.103.6 | 443:30993/TCP | 52s |

Now let's test the STS with the following `curl` command run from your local machine. This is exactly the same `curl` command we used in chapter 7 in section 7.6. The IP address in the command is the external IP address corresponding to the `sts-service` from the previous command.

```
|> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo"
https://34.82.103.6/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret (which are hardcoded in the STS). If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise).

```
{"access_token":"eyJhbGciOiJSUzI1NiIsInRcIiKpXVCJ9.eyJleHAiOiE1NTEzMTIzNzYsInVzZXJfbmFtZSI6InBldGVyliwiYXV0aG9yaXRpZXMiOlsUk9MRV9VU0VSII0sImp0aSi6ljRkMmjNjQ4LTQ2MWQtNGVIYy1hZTljLTVIYWyUxZjA4ZTjhMiSlmNsawVvudF9pZC16ImFwcGxpY2F0aW9uaWQiLCjzY29wZSI6WyJmb28iXX0.tr4yUmGLtsH7q9Gei2i7gxyTsOOa0RS0Yoc2uBuAW5OVIKZcVsITWV3bDN0FVBzimpAPy33tvicFROhBFoVThqKzzG005kURN5bnQ4uFLAP0Npz26BuDjVmwXNxRQp2lVXl4lQ4eTvuyZojUSCxzCl1LNw5EFFi22J73g1_mRm2j-dEhBp1TvMaRKLBDk2hzIDVKzu5oj_gODBFm3a1S-IijYoCimIm2igcesXkhipRjtjNcrJSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97jlT6Rr1XZBLveyjfKs_XIxgU5qizRm9mt5xg","token_type":"bearer","refresh_token":"","expires_in":5999,"scope":"foo","jti":"4d2bb648-461d-4eec-ae9c-5sea1f08e2a2"}
```

Here we talked to the STS running in Kubernetes over Transport Layer Security (TLS). The STS uses TLS certificates embedded into the `prabath/insecure-sts-ch10:v1` Docker image, and the Kubernetes load balancer just tunnels³ all the requests it gets to the corresponding container.

11.2 Managing secrets in a Kubernetes environment

In section 11.1 we used a Docker image called `prabath/insecure-sts-ch10:v1`. We named it `insecure-sts` for a reason. In chapter 10.2 we had a detailed discussion on why this image is insecure. While creating this image we embedded all the keys and the credentials to access the keys to the image itself. Because this is in the Docker Hub, anyone having access to the image

³In addition to TLS tunneling, we can also do TLS termination at the Kubernetes load balancer. Then a new connection is created between the load balancer and the corresponding service.

can figure out all our secrets, and it's the end of the world! You can find the source code of this insecure STS in the chapter10/sample01 directory.

To make the Docker image secure, the first thing we need to do is to externalize all the keystores and credentials. In chapter 10.2.2 we discussed how to externalize the application.properties file (where we keep all the credentials) from the Docker image and also the two keystore files (one keystore includes the key to secure the TLS communication, while the other keystore includes the key to sign JWT access tokens the STS issues). We published this updated Docker image to the Docker Hub as prabath/secure-sts-ch10:v1. To help you understand how this Docker image is built, listing 11.3 repeats the Dockerfile from chapter 10, section 10.2.2.

Listing 11.3. The content of the Dockerfile used to build the secure STS

```
FROM openjdk:8-jdk-alpine
ADD target/com.manning.mss.ch10.sample01-1.0.0.jar com.manning.mss.ch10.sample01-1.0.0.jar
ENV SPRING_CONFIG_LOCATION=/opt/application.properties
ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
```

Listing 11.3 shows that we've externalized the application.properties file. In other words, Spring Boot reads the location of the application.properties file from the SPRING_CONFIG_LOCATION environment variable, which is set to /opt/application.properties. So Spring Boot expects the application.properties file to be present in the /opt directory of the Docker container. Because our expectation here is to externalize the application.properties file, we can't put it to the container file system.

In chapter 10.2.2 we used Docker volume mounts, so Docker loads the application.properties file from the host machine and maps it to the /opt directory of the container file system. Following is the command we used in chapter 10.2.2 to run the Docker container with volume mounts (only for your reference; if you want to try it, follow the instructions in chapter 10, section 10.2.2).

```
> docker run -p 8443:8443 --mount type=bind,source="$(pwd)/keystores/keystore.jks,target=/opt/keystore.jks" --mount type=bind,source="$(pwd)/keystores/jwt.jks,target=/opt/jwt.jks" --mount type=bind,source="$(pwd)/config/application.properties,target=/application.properties" -e KEYSTORE_SECRET=springboot -e JWT_KEYSTORE_SECRET=prabath/secure-sts-ch10:v1
```

In the command, we use volume mounts not only to pass the application.properties file, but also the two keystore files. If you look at the keystore locations mentioned in the application.properties file (listing 11.4), Spring Boot looks for the keystore.jks and jwt.jks files inside the /opt directory of the container file system. Also, in listing 11.4, you can see that we've externalized the keystore passwords. Now, Spring Boot reads the password of the keystore.jks file from the KEYSTORE_SECRET environment variable and the password of the jwt.jks file from the JWT_KEYSTORE_SECRET environment variable, which we pass in the docker run command.

Listing 11.4. The content of the application.properties file

```
server.port: 8443
server.ssl.key-store: /opt/keystore.jks
server.ssl.key-store-password: ${KEYSTORE_SECRET}
server.ssl.keyAlias: spring
spring.security.oauth.jwt: true
spring.security.oauth.jwt.keystore.password: ${JWT_KEYSTORE_SECRET}
spring.security.oauth.jwt.keystore.alias: jwtkey
spring.security.oauth.jwt.keystore.name: /opt/jwt.jks
```

11.2.1 Using ConfigMap to externalize configurations in Kubernetes

When you run a container in a Kubernetes environment, you can't pass configuration files from your local file system like we did with Docker in section 11.2.0. Kubernetes introduces an object called `ConfigMap` to decouple configuration from containers or microservices running in a Kubernetes environment. In this section, you'll learn how to represent the `application.properties` file, the `keystore.jks` file, the `jwt.jks` file, and the keystore passwords as `ConfigMap` objects.

A `ConfigMap` is not the ideal object to represent sensitive data like keystore passwords. In such cases, we use another Kubernetes object called `Secret`. In section 11.3 we'll move keystore passwords from a `ConfigMap` to a Kubernetes `Secret`. If you're new to Kubernetes `ConfigMaps`, see appendix B for the details and find out how it works internally.

11.2.2 Defining a ConfigMap for application.properties

Kubernetes lets you create a `ConfigMap` object with the complete content of a configuration file. Listing 11.5 shows the content of the `application.properties` file under the `data` element in the `application.properties` key. The name of the key must match the name of the file we expect to be in the container file system. You can find the complete `ConfigMap` definition of the `application.properties` file in the `chapter11/sample01/sts.configuration.yaml` file.

Listing 11.5. The definition of application-properties-config-map

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sts-application-properties-config-map
data: #A
  application.properties: | #B
    [
      server.port: 8443
      server.ssl.key-store: /opt/keystore.jks
      server.ssl.key-store-password: ${KEYSTORE_SECRET}
      server.ssl.keyAlias: spring
      spring.security.oauth.jwt: true
      spring.security.oauth.jwt.keystore.password: ${JWT_KEYSTORE_SECRET}
      spring.security.oauth.jwt.keystore.alias: jwtkey
      spring.security.oauth.jwt.keystore.name: /opt/jwt.jks
    ]
```

#A Creates a ConfigMap object of a file with a text representation

#B The name of the key must match the name of the file we expect to be in the container file system.

Once we define the ConfigMap in a yaml file, we can use the kubectl client to create an actual ConfigMap object in the Kubernetes environment. We differ that until section 11.2.5, when we complete our discussion on the other three ConfigMap objects as well (in sections 11.2.3 and 11.2.4).

11.2.3 Defining ConfigMaps for keystore.jks and jwt.jks files

Kubernetes lets you create a ConfigMap object of a file with a text representation (listing 11.5) or with a binary representation. In listing 11.6 we use the binary representation option to create ConfigMaps for the keystore.jks and jwt.jks files. The base64-encoded content of the keystore.jks file is listed under the key `keystore.jks` under the element `binaryData`. The name of the key must match the name of the file we expect to be in the /opt directory of the container file system.

You can find the complete ConfigMap definition of the keystore.jks and jwt.jks files in the chapter11/sample01/sts.configuration.yaml file. Also, the keystore.jks and jwt.jks binary files are available in the chapter10/sample01/keystores directory in case you'd like to do file-to-base64 conversion yourself.⁴

Listing 11.6. The definition of ConfigMap for keystore.jks and jwt.jks

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sts-keystore-config-map
binaryData: #A
  keystore.jks: [base64-encoded-text]
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: sts-jwt-keystore-config-map
binaryData:
  jwt.jks: [base64-encoded-text]
```

#A Creates a ConfigMap object of a file with a binary representation

11.2.4 Defining a ConfigMap for keystore credentials

First, *don't do this in a production deployment!* Kubernetes stores anything that you store in a ConfigMap in cleartext. To store credentials in a Kubernetes deployment, we use a Kubernetes object called Secret instead of a ConfigMap. We talk about Secrets later in section 11.3. Until then, we'll define keystore credentials as a ConfigMap.

⁴To convert a binary file to a base64-encoded text file, you can use an online tool like <https://www.browserling.com/tools/file-to-base64>.

Listing 11.7 shows the definition of the `sts-keystore-credentials` ConfigMap. There we pass the password to access the `keystore.jks` file under the `KEYSTORE_PASSWORD` key and the password to access the `jwt.jks` file under the `JWT_KEYSTORE_PASSWORD` key, both under the `data` element. You can find the complete ConfigMap definition of keystore credentials in the `chapter11/sample01/sts.configuration.yaml` file.

Listing 11.7. The definition of keystore credentials

```
apiVersion: v1
  apiVersion: v1
kind: ConfigMap
metadata:
  name: sts-keystore-credentials
data:
  KEYSTORE_PASSWORD: springboot
  JWT_KEYSTORE_PASSWORD: springboot
```

11.2.5 Creating ConfigMaps using the kubectl client

In the file `chapter11/sample01/sts.configuration.yaml`, you'll find ConfigMap definitions of all four ConfigMaps we've discussed in this section thus far. You can use the following `kubectl` command from the `chapter11/sample01` directory to create ConfigMap objects in your Kubernetes deployment:

```
\> kubectl apply -f sts.configuration.yaml
configmap/sts-application-properties-config-map created
configmap/sts-keystore-config-map created
configmap/sts-jwt-keystore-config-map created
configmap/sts-keystore-credentials created
```

The following `kubectl` command lists all the ConfigMap objects available in your Kubernetes cluster (under the current namespace):

```
\> kubectl get configmaps
NAME           DATA   AGE
sts-application-properties-config-map   1    50s
sts-keystore-config-map      0    50s
sts-jwt-keystore-config-map     0    50s
sts-keystore-credentials       2    50s
```

11.2.6 Consuming ConfigMaps from a Kubernetes deployment

In this section we'll go through the changes we need to introduce to the Kubernetes deployment that we created in listing 11.1 to read the values from the ConfigMaps we created in section 11.2.5. You'll find the complete updated definition of the Kubernetes deployment in the `chapter11/sample01/sts.deployment.with.configmap.yaml` file.

In this section we'll focus on two types of ConfigMaps: one is where we want to read the content of a file from a ConfigMap and mount that file into the container file system, and the other one is where we want to read a value from a ConfigMap and set that as an environment

variable in the container. Listing 11.8 shows part of the deployment object that carries the configuration related to the containers.

Listing 11.8. Part of the security token service deployment definition

```
containers:  
- name: sts  
  image: prabath/secure-sts-ch10:v1  
  imagePullPolicy: Always  
  ports:  
    - containerPort: 8443  
  volumeMounts: #A  
    - name: application-properties #B  
      mountPath: "/opt/application-properties" #C  
      subPath: "application-properties"  
    - name: keystore  
      mountPath: "/opt/keystore.jks"  
      subPath: "keystore.jks"  
    - name: jwt-keystore  
      mountPath: "/opt/jwt.jks"  
      subPath: "jwt.jks"  
  env: #D  
    - name: KEYSTORE_SECRET #E  
      valueFrom:  
        configMapKeyRef:  
          name: keystore-credentials #F  
          key: KEY_STORE_PASSWORD #G  
    - name: JWT_KEYSTORE_SECRET  
      valueFrom:  
        configMapKeyRef:  
          name: keystore-credentials  
          key: KEY_STORE_PASSWORD  
  volumes:  
    - name: application-properties #H  
      configMap:  
        name: sts-application-properties-config-map #I  
    - name: keystore  
      configMap:  
        name: sts-keystore-config-map  
    - name: jwt-keystore  
      configMap:  
        name: sts-jwt-keystore-config-map
```

#A Defines the volume mounts used by this Kubernetes deployment

#B Names the volume, which refers to the volumes section

#C Location of the container file system to mount this volume

#D Defines the set of environment variables read by the Kubernetes deployment

#E The name of the environment variable. This is the exact name you find in application.properties.

#F The name of the ConfigMap to read the environment variable

#G Reads the environment variable from the corresponding ConfigMap

#H Names the volume, which is referred by the name element of the volumeMounts section

#I Names the ConfigMap, which carries the data related to application.properties

You can use the following `kubectl` command from the chapter11/sample01 directory to update the Kubernetes deployment with the changes highlighted in listing 11.8:

```
\> kubectl apply -f sts.deployment.with.configmap.yaml
deployment.apps/sts-deployment configured
```

The Kubernetes service we created in section 11.1.4 requires no changes. Make sure it's up and running with the correct IP address using the `kubectl get services` command. Now let's test the STS with the following `curl` command run from your local machine.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo"
https://34.82.103.6/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise).

```
{"access_token":"eyJhbGciOiJSUzI1NlslnR5cCl6IkpxVCVCI9.yJleHAIoJE1NTezMTIzNzYslnVzZXJfbmFtZSI6InBldGVyliwiYXV0aG9y
aXRpZXMiOlsIuk9MRV9VU0VSII0sImp0aSl6jRkMmlnJQ4LTQ2MWQtNGVYy1hTljlTVlYWUxZjA4ZTjhMilsImNsawVvudF9pZCI
6lmFwcGxpY2F0aW9uaWQiLCjzY29wZSI6WyJmb28IXX0.tr4yUmGLtsH7q9Ge2i7gxyTsOOa0RS0Yoc2uBuAW5OVIKzCvslTWV3
bDN0FVHBzimpAPy33ticFROhBFoVThqKXzzG005kURN5bnQ4uFLAP0Npz26BuDjvVmwxNxrQp2lVXl4IQ4eTvuyZozjUSCXzCl1LN
w5EFFi22J73g1_mRm2j-dEhBp1TvMaRKLBDk2hzlDVKzu5oj_gODBFm3a1-
IjjYoCimlmlm2igcesXkipRJtjNcrJSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97jIT6Rr1XZBlveyjfKs_
XlXgU5qizRm9mt5xg","token_type":"bearer","refresh_token":"","expires_in":5999,"scope":"foo","jti":"4d2bb648-461d-4eec-
ae9c-5cae1f08e2a2"}
```

11.2.7 Loading keystores with an init container

In a Kubernetes pod, we can run more than one container in a pod, but as a practice, we only run one application container. Along with an application container, we can also run one or more init containers. If you're familiar with Java (or any other programming language), an init container in Kubernetes is like a constructor in a Java class. As the constructor in a Java class runs well before any other methods, an init container in a pod must run and complete before any other application containers in the pod. This is a great way to initialize a Kubernetes pod. You can pull any files (keystore, policies, and so forth), configurations, and so on with an init container.

Listing 11.9 modifies the STS deployment to load `keystore.jks` and `jwt.jks` files from the Git repository using an init container instead of loading it from a config file. You can find the complete updated definition of the Kubernetes deployment in the `chapter11/sample01/sts.deployment.with.init.containers.yaml` file. Listing 11.9 shows part of the updated STS deployment, corresponding to the init container.

Listing 11.9. The STS deployment with an init container

```
initContainers: #A
- name: init-keystores
image: busybox:1.28 #B
command: #C
- "/bin/sh"
- "-c"
- "wget https://github.com/microservices-security-in-action/samples/raw/master/chapter10/sample01/keystores/jwt.jks -O /opt/jwt.jks | wget https://github.com/microservices-security-in-
```

```

action/samples/raw/master/chapter10/sample01/keystores/keystore.jks -O /tmp/keystore.jks"
volumeMounts: #D
- name: keystore #E
  mountPath: "/opt/keystore.jks" #F
  subPath: "keystore.jks" #G
- name: jwt-keystore
  mountPath: "/opt/jwt.jks"
  subPath: "jwt.jks"

```

In listing 11.9 we created a container with the busybox Docker image. Because the busybox container is configured as an init container, it runs before any other container in the pod. Under the `command` element (also in listing 11.9), we specified the program the busybox container should run. There we got both `keystore.jks` and `jwt.jks` files from a Git repo and copied both `keystore.jks` and `jwt.jks` to the `/opt` directory of the busybox container file system.

The whole objective of the init container is to get the two keystores into the Docker container that runs the STS. To do that, we need to have two volume mounts, where both the volumes (with the name `keystore` and `jwt-keystore`) are mapped to the `/opt` directory. Because we already have volume mounts with these two names (under the `secure-sts` container in listing 11.10), the two keystores are also visible to the `secure-sts` container file system.

Listing 11.10. Volume mounts in secure-sts container

```

volumeMounts:
- name: application-properties
  mountPath: "/opt/application.properties"
  subPath: "application.properties"
- name: keystore
  mountPath: "/opt/keystore.jks"
  subPath: "keystore.jks"
- name: jwt-keystore
  mountPath: "/opt/jwt.jks"
  subPath: "jwt.jks"

```

Finally, to support init containers, we also need to do one more change to the original STS deployment. Earlier, under the `volumes` element of the STS deployment, we pointed to the corresponding ConfigMaps, and now we need to point it to an empty directory as shown here:

```

volumes:
- name: application-properties
  configMap:
    name: sts-application-properties-config-map
- name: keystore
  emptyDir: {}
- name: jwt-keystore
  emptyDir: {}

```

Let's use the following `kubectl` command with the `chapter11/sample01/sts.deployment.with.init.containers.yaml` file to update the STS deployment to use init containers:

```
\> kubectl apply -f sts.deployment.with.initcontainer.yaml  
deployment.apps/sts-deployment configured
```

11.3 Kubernetes secrets

As we discussed in section 11.2.4, the ConfigMap is not the right way of externalizing sensitive data in Kubernetes. A Secret is a Kubernetes object, just like a ConfigMap, which carries name/value pairs, but is ideal for storing sensitive data. In this section we discuss Kubernetes Secrets in detail and see how we can update the security token service deployment with Kubernetes Secrets, instead of using ConfigMaps, to externalize keystore credentials.

11.3.1 The default token secret in every container

Kubernetes provisions a Secret to each container of the pod it creates. This is called the default token secret. To see the default Secret, run the following `kubectl` command:

```
\> kubectl get secrets  
NAME          TYPE           DATA  AGE  
default-token-l9fj8  kubernetes.io/service-account-token  3    10d
```

Listing 11.11 shows the structure of the Secret returned by `kubectl` in yaml format. In that listing, the name/value pairs under the `data` element carries the confidential data in base64-encoded format. The default token secret has three name value pairs: `ca.crt`, `namespace`, and `token`. The listing here only shows part of the values for `ca.crt` and `token`.

Listing 11.11. The default Kubernetes secret

```
\> kubectl get secret default-token-l9fj8 -o yaml  
apiVersion: v1  
kind: Secret  
metadata:  
  annotations:  
    kubernetes.io/service-account.name: default  
    kubernetes.io/service-account.uid: ff3d13ba-d8ee-11e9-a88f-42010a8a01e4  
  name: default-token-l9fj8  
  namespace: default  
  type: kubernetes.io/service-account-token  
data:  
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ...  
  namespace: ZGVmYXVsdA==  
  token: ZXlKaGJHY2lPaUpTVXpJMUs...  
  
```

The value of `ca.crt` is, in fact, the root certificate of the Kubernetes cluster. You can use an online tool like <https://base64.guru/converter/decode/file> to convert base64-encoded text to a file. You'll see something similar to the following, which is the PEM-encoded root certificate of the Kubernetes cluster.

```
-----BEGIN CERTIFICATE----- MIIDCzCCAfOgAwIBAgIQdzQ6I91oRfLI141a9hEPoTANBgkqhkiG9w0BAQsFADAv  
MS0wKwYDVQQDEyRkmwJjZGU1MC1jNjNkLTQ5MWYtOTZlNi0wNTEwZDliOTI5ZTEw  
HhcNMTkwOTE3MDA1ODI2WhcNMjQwOTE1MDE1ODI2WjAvMS0wKwYDVQQDEyRkmwJj  
ZGU1MC1jNjNkLTQ5MWYtOTZlNi0wNTEwZDliOTI5ZTEwggEiMA0GCSqGSIb3DQE...  
  
```

```

AQUAA4IBDwAwggEKAoIBAQChdg15gwelqZzraHBFH3sB9FKfv2IDZ03/MAq6ek3J
Njj+7huiUy6Pu9t5rOjGU/JivRi7Xipqc/JGMRjmMVwCmSv6D+5N8+jmvhZ4i
uzbjUOpuyozRsrmf3hzbwbcLbcA94Y1d+oKOTZ+IYs8XNhXORCM+gDKryC5MeGnY
zqd+/MLS6zajG3qlGQAWn9XKCIpPjRDOjh5h/uNQs+r2Y9Uz4oi4shVUVxibwOhrh
0MpAt6BGujDMNDNRGH8/dK1CZ1EYjYoUaOT0eF21RSJ2y82AF55eA17hSxY4j6x5
3ipQt1pe49j5m7QU5s/VoDGsBBge6vYd0AU9y96xFUvAgMBAAGjizAhMA4GA1Ud
DwEB/wQEAwICBDAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSib3DQEBCwUA4IBAQB4
33lsGOSU2z6PKLdnZHrnnwZq44AH3CzCQ+M6cQPTU63XHXwCEQtxSDcjDTm1xZqR
qeoUcgCW4mBjdG4dMkQD+MuBu0oGLQPkv5XsnJg+4zRhKTd78PUE15ZF8HBX5Vt
+3lrbElvhREuwDGCPmMRO/081ZlwLZFrRwRAZQmkEgCtfcOUG03+HLQw1U2P
xKFLx6ISUNSkPf05pkBW6Tg3JfQnfuKUPxUFI/3JuJxDzl2XLx7GFF1J4tW812A
T6WfgDvYS2Ld9o/rw3C036NtvdjGrnb2QqEosGeDPQOXs5sgFT8LPNkQ+f/8nn G0Jk4TNzdxezmyyyvxh2
-----END CERTIFICATE-----

```

To get something meaningful out of this, you can use an online tool like https://report-uri.com/home/pem_decoder to decode the PEM file, resulting in something similar to the following:

```

Common Name: d1bcde50-c63d-491f-96e6-0510d9b929e1
Issuing Certificate: d1bcde50-c63d-491f-96e6-0510d9b929e1
Serial Number: 77343A97DD6845F2C8D78D5AF6110FA1
Signature: sha256WithRSAEncryption
Valid From: 00:58:26 17 Sep 2019
Valid To: 01:58:26 15 Sep 2024
Key Usage: Certificate Sign
Basic Constraints: CA:TRUE

```

The token under the `data` element in listing 11.11 carries a JSON Web Token (see appendix H for details on JWT). This JWT is itself base64-encoded. You can use an online tool like <https://www.base64decode.org> to base64-decode the token and an online JWT decoder like <http://jwt.io> to decode the JWT. The following shows the decoded payload of the JWT:

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-l9fj8",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "ff3d13ba-d8ee-11e9-a88f-42010a8a01e4",
  "sub": "system:serviceaccount:default:default"
}
```

Each container in a Kubernetes pod has access to this JWT from the `/var/run/secrets/kubernetes.io/serviceaccount` directory, in its own container file system. If you want to access the Kubernetes API server from a container, then you can use this JWT for authentication. In fact, this JWT is bound to a Kubernetes service account. We discuss service accounts in detail in section 11.6.

11.3.2 Updating the STS to use Secrets

In section 11.2 we updated the STS deployment to use ConfigMaps to externalize configuration data. Even for keystore credentials, we used ConfigMaps instead of Secrets. In this section, we're going to update the STS deployment to use Secrets to represent keystore credentials.

First, we need to define the `Secret` object as shown in listing 11.12. The complete definition of the `Secret` object is in the `chapter11/sample01/sts.secrets.yaml` file.

Listing 11.12. The definition of the `Secret` object that carries keystore credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: sts-keystore-secrets
stringData:
  KEYSTORE_PASSWORD: springboot
  JWT_KEYSTORE_PASSWORD: springboot
```

To create the `Secret` in the Kubernetes environment, use the following command from `chapter11/sample01` directory:

```
\> kubectl apply -f sts.secrets.yaml
secret/sts-keystore-secrets created
```

In listing 11.12 we defined keystore credentials under the `stringData` element. Another option is to define credentials under the `data` element. In listing 11.16 (later in the chapter), we have an example for that.

When you define credentials under the `data` element, you need to base64-encode the values. If you mostly use binary credentials like private keys, we need to use the `data` element. For text credentials, the `stringData` element is the preferred option. Another important thing to notice is that Kubernetes has designed the `stringData` element to be write-only. That means, when you try to view a `Secret` you defined with `stringData`, it won't return as a `stringData` element, rather Kubernetes base64-encodes the values and returns those under the `data` element. You can use the following `kubectl` command to list the definition of the `Secret` object we created in listing 11.12 in yaml format:

```
\> kubectl get secret sts-keystore-secrets -o yaml
apiVersion: v1
kind: Secret
metadata:
  name: sts-keystore-secrets
data:
  KEYSTORE_PASSWORD: c3ByaW5nYm9vdA==
  JWT_KEYSTORE_PASSWORD: c3ByaW5nYm9vdA==
```

Now let's see how to update the STS deployment to use the `Secret` object we created. You can find the updated yaml configuration for the STS deployment in the `chapter11/sample01/sts.deployment.with.secrets.yaml` file. Listing 11.13 shows part of the complete STS deployment, which reads keystore credentials from the `Secret` object and populates the environment variables.

Listing 11.13. Part of the STS deployment definition using secrets

```
env:
  - name: KEYSTORE_SECRET
```

```
valueFrom:  
secretKeyRef:  
  name: sts-keystore-secrets  
  key: KEYSTORE_PASSWORD  
- name: JWT_KEYSTORE_SECRET  
valueFrom:  
secretKeyRef:  
  name: sts-keystore-secrets  
  key: JWT_KEYSTORE_PASSWORD
```

Let's run the following `kubectl` command from `chapter11/sample01` to update the STS deployment:

```
\> kubectl apply -f sts.deployment.with.secrets.yaml  
deployment.apps/sts-deployment configured
```

11.3.3 How does Kubernetes store Secrets?

The reason you have to pick Secrets over ConfigMaps to store sensitive data relies on how Kubernetes internally handles Secrets. Kubernetes makes sure that the sensitive data Kubernetes represents as Secrets are only accessible to the pods that need them, and even in such cases, none of the Secrets are written to disk, but only kept in memory. The only place Kubernetes writes Secrets to disk is at the master node, where all the Secrets are stored in etcd (see appendix B), which is the Kubernetes' distributed key/value store. From the Kubernetes 1.7 release onwards, etcd stores Secrets only in an encrypted format.

11.4 Running the Order Processing microservice in Kubernetes

In this section we're going to deploy the Order Processing microservice in Kubernetes. As in figure 11.1, the Order Processing microservice trusts the tokens issued by the STS, which we now have running in Kubernetes. Once the client application passes the JWT to the Order Processing microservice, the Order Processing microservice talks to the STS to retrieve its public key to validate the signature of the JWT. This is the only communication that happens between the Order Processing microservice and the STS. In fact, to be precise, the Order Processing microservice doesn't wait until it gets a request to talk to the STS, rather it talks to the STS at start up to get its public key.

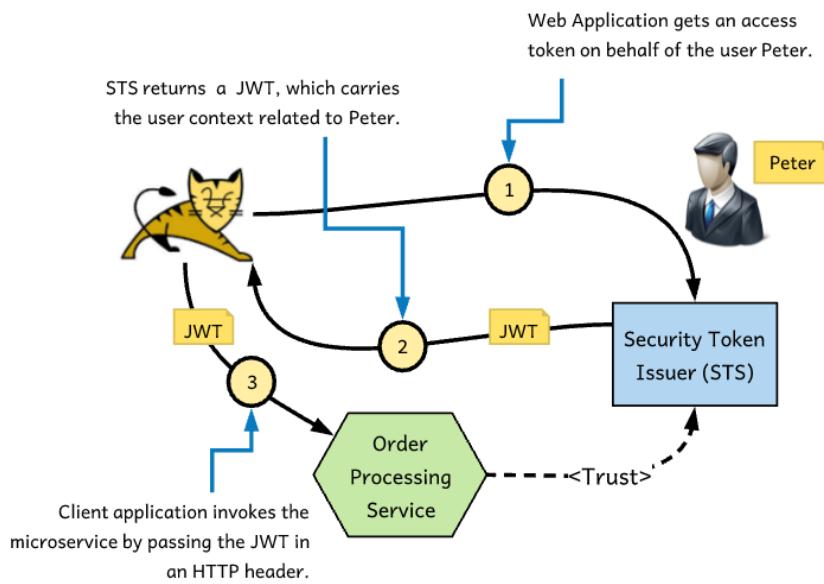


Figure 11.1 An STS issues a JWT access token to the web application, and the web application uses it to access the microservice on behalf of the user, Peter.

In chapter 10, under section 10.4, we explained how to run the Order Processing microservice as a Docker container. This is the Docker command we used in section 10.4, which externalized the application.properties file, the keystore (keystore.jks), the trust store (trust-store.jks), the keystore credentials, and the trust store credentials. You don't need to run this command now; if you want to try it out, follow the instructions in chapter 10.

```
> docker run --net manning-network -p 9443:9443 --mount
  type=bind,source="${pwd}/keystores/keystore.jks,target=/opt/keystore.jks" --mount
  type=bind,source="${pwd}/keystores/trust-store.jks,target=/opt/trust-store.jks" -e KEYSTORE_SECRET=springboot -
  e TRUSTSTORE_SECRET=springboot --mount
  type=bind,source="${pwd}/config/application.properties,target=/opt/application.properties" prabath/order-
  processing:v1
```

To deploy the Order Processing microservice in Kubernetes, we need to create a Kubernetes deployment and a service. This is similar to what we did before when deploying the STS in Kubernetes.

11.4.1 Creating ConfigMaps/Secrets for the Order Processing microservice

In this section we create three ConfigMaps to externalize application.properties and two keystores (keystore.jks and trust-store.jks) and a Secret to externalize the keystore credentials. Listing 11.14 shows the definition of the ConfigMap for the application.properties file. The value

of security.oauth2.resource.jwt.key-uri in listing 11.14 carries the endpoint of the STS. Here the sts-service hostname is the name of Kubernetes service we created for the STS.

Listing 11.14. The definition of the application.properties ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: orders-application-properties-config-map
data:
  application.properties: |
    [
      server.port: 8443
      server.ssl.key-store: /opt/keystore.jks
      server.ssl.key-store-password: ${KEYSTORE_SECRET}
      server.ssl.keyAlias: spring
      server.ssl.trust-store: /opt/trust-store.jks
      server.ssl.trust-store-password: ${TRUSTSTORE_SECRET}
      security.oauth2.resource.jwt.key-uri: https://sts-service/oauth/token_key
      inventory.service: https://inventory-service/inventory
      logging.level.org.springframework=DEBUG
      logging.level.root=DEBUG
    ]
```

Listing 11.15 shows the ConfigMap definition for the keystore.jks and trust-store.jks files. Each binaryData element in each ConfigMap definition in listing 11.15 carries the base64-encoded text of the corresponding keystore file.

Listing 11.15. The definition of the keystore ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: orders-keystore-config-map
binaryData:
  keystore.jks: [base64-encoded-text]
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: orders-truststore-config-map
binaryData:
  trust-store.jks: [base64-encoded-text]
```

Listing 11.16 shows the Secret definition for the credentials of in the keystore.jks and trust-store.jks files. The value of each key under the data element in listing 11.16 carries the base64-encoded text of corresponding credentials. You can use the following command on a Mac terminal to generate the base64encoded value of a given text:

```
\> echo -n "springboot" | base64
c3ByaW5nYm9vdA==
```

Listing 11.16. The definition of the keystore credentials Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: orders-key-credentials
type: Opaque
data:
  KEYSTORE_PASSWORD: c3ByaW5nYm9vdA==
  TRUSTSTORE_PASSWORD: c3ByaW5nYm9vdA==
```

In the chapter11/sample02/order.processing.configuration.yaml file, you'll find ConfigMap and Secret definitions of all that we discussed in this section. You can use the following `kubectl` command from the chapter11/sample02 directory to create ConfigMap and Secret objects in your Kubernetes deployment:

```
\> kubectl apply -f order.processing.configuration.yaml
configmap/orders-application-properties-config-map created
configmap/orders-keystore-config-map created
configmap/orders-truststore-config-map created
secret/orders-key-credentials created
```

The following two `kubectl` commands list all the ConfigMap and Secret objects available in your Kubernetes cluster (under the current namespace):

```
\> kubectl get configmaps
NAME           DATA   AGE
orders-application-properties-config-map 1    50s
orders-keystore-config-map      0    50s
orders-truststore-config-map     0    50s

\> kubectl get secrets
NAME           DATA   AGE
orders-key-credentials     2    50s
```

11.4.2 Creating a deployment for the Order Processing microservice

In this section we create a deployment in Kubernetes for the Order Processing microservice that we defined in the yaml file found in the chapter11/sample02/order.processing.deployment.yaml directory. You can use the following `kubectl` command from the chapter11/sample02 directory to create the Kubernetes deployment:

```
\>kubectl apply -f order.processing.deployment.with.configmap.yaml
deployment.apps/orders-deployment created
```

11.4.3 Creating a service for the Order Processing microservice

To expose the Kubernetes deployment we created in section 11.4.2 for the Order Processing microservice, we also need to create a Kubernetes service. You can find the definition of this service in the yaml file in the chapter11/sample02/order.processing.service.yaml file. Use the

following `kubectl` command from the `chapter11/sample02` directory to create the Kubernetes service:

```
\> kubectl apply -f order.processing.service.yml  
service/orders-service created
```

Then use the following command to find all the services in your Kubernetes cluster (under the current namespace). It takes a few minutes for Kubernetes to assign an external IP address for the order service we just created. If you run the same command after couple of minutes, you'll notice the following output with an external IP address assigned to the service:

```
\> kubectl get services  
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)      AGE  
kubernetes  ClusterIP  10.39.240.1 <none>        443/T      5d21h  
orders-service  LoadBalancer  10.39.249.66  35.247.11.161  443:32401/TCP  72s  
sts-service  LoadBalancer  10.39.255.168 34.83.188.72  443:31749/TCP  8m39s
```

11.4.4 Testing the end-to-end flow

In this section we test the end-to-end flow (figure 11.2). First we need to get a token from the STS and then use it to access the Order Processing microservice. Now we've got both services running on Kubernetes. Let's use the following `curl` command, run from your local machine, to a get a token from the STS. Make sure you use the correct external IP address of the STS.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo"  
https://34.83.188.72/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cI6IkpxVCJ9eyJleHAiOjE1NTEzMThzNzYsInVzZXJfbmFtZSI6InBldGVyliwiYXV0aG9yaXRpZXMiOlsUiUk9MRV9VU0VSII0sImp0aSi6ljRkMmJiNjQ4LTQ2MWQtNGVlYy1hZTljLTVlYWUxZjA4ZTjhMiIsImNsawVvudF9pZC16lmFwcGxpY2F0aW9uaWQiLCjzY29wZSI6WyJmb28iXX0.tr4yUmGLtsH7q9Ge2i7gxyTsOOaORS0Yoc2uBuAW5OVIKZcVsITWV3bDNOFVHBzimPApy33tvicFROhBFoVThqKXzzG00SkURN5bnQ4uFLAP0Npz6BuDjVmwxNxRQp2lVxl4lQ4eTvuyZozjUSCxzCl1LNw5EFFi22J73g1_mRm2j-dEhBp1TvMaRKLBDk2hzlDVKzu5oj_gODBFm3a1S-IijYoCimlm2igcesXkhipRjtjNcrJSegBbGgyXHVak2gB7l07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97jIT6Rr1XZB1veyjfKs_XIXgU5qizRm9mt5xg","token_type":"bearer","refresh_token":"","expires_in":5999,"scope":"foo","jti":"4d2bb648-461d-4eec-aec9-5eae1f08e2a2"}
```

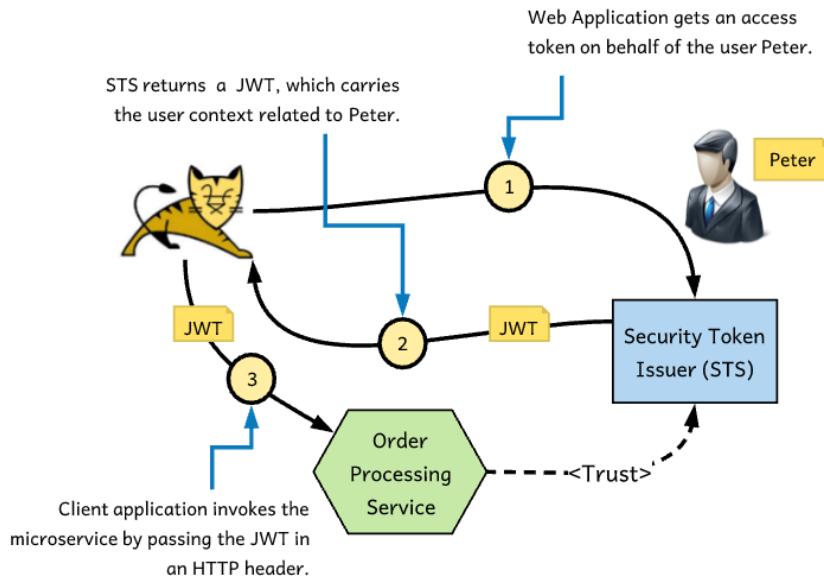


Figure 11.2 The STS issues a JWT access token to the web application, and the web application uses it to access the microservice on behalf of the user, Peter.

Now try to invoke the Order Processing microservice with the JWT you got from the previous `curl` command. Set the same JWT we got in the HTTP Authorization Bearer header, using the following `curl` command and invoke the Order Processing microservice. Because the JWT is a little lengthy, you can use a small trick when using the `curl` command in this case. Export the JWT to an environmental variable (`TOKEN`) and then use that environmental variable in your request to the Order Processing microservice as shown here:

```
> export TOKEN=jwt_access_token
> curl -k -H "Authorization: Bearer $TOKEN" https://35.247.11.161/orders/11
{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}
```

11.5 Running the Inventory microservice in Kubernetes

In this section we introduce another microservice, the Inventory microservice, to our Kubernetes deployment and see how service-to-service communication works (figure 11.3). Here, when you invoke the Order Processing microservice with a JWT token obtained from the STS, the Order Processing microservice internally talks to the Inventory microservice.

Because the process of deploying the Inventory microservice on Kubernetes is similar to the process we followed while deploying the Order Processing microservice, we won't go into details. The only key difference is that the Kubernetes service corresponding to the Inventory

microservice is of ClusterIP type (or the default service type) because we don't want external client applications to directly access it.

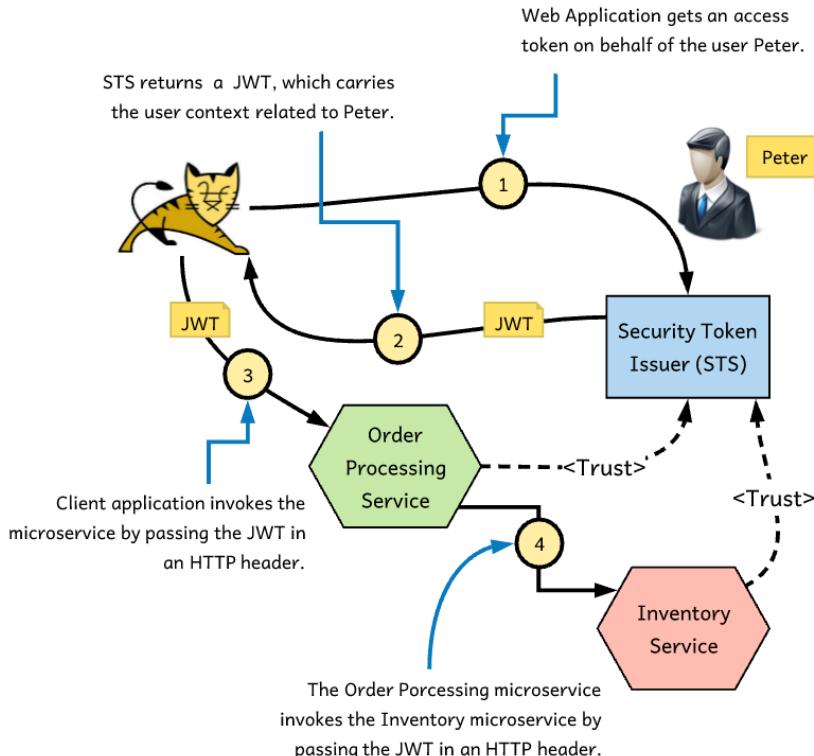


Figure 11.3 STS issues a JWT access token to the web application and the web application uses it to access the microservice on behalf of the user, Peter. The Order Processing microservice use the same JWT it got from the client application to access the Inventory microservice.

Let's run the following `kubectl` command from the `chapter11/sample03` directory to create a Kubernetes deployment for the Inventory microservice. This command creates a set of ConfigMaps, a Secret, a Deployment, and a Service:

```
\> kubectl apply -f.  
configmap/inventory-application-properties-config-map created  
configmap/inventory-keystore-config-map created  
configmap/inventory-truststore-config-map created  
secret/inventory-key-credentials created  
deployment.apps/inventory-deployment created  
service/inventory-service created
```

Use the following command to find all the services in your Kubernetes cluster (under the current namespace). Because the Inventory service is a service of `ClusterIP` type, you won't find an external IP address for it.

```
\> kubectl get services
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)
inventory-service ClusterIP  10.39.251.182 <none>        443/TCP
orders-service   LoadBalancer 10.39.245.40  35.247.11.161  443:32078/TCP
sts-service     LoadBalancer 10.39.252.24  34.83.188.72  443:30288/TCP
```

Let's test the end-to-end flow (figure 11.3). First, we need to get a token from the STS and then use it to access the Order Processing microservice. Now we've got both services running on Kubernetes. Let's use the following `curl` command, run from your local machine, to a get a token from the STS. Make sure you use the correct external IP address of the STS.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo"
https://34.83.188.72/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works, the STS returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise).

```
{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9eyJleHAiOiE1NTEzMTIzMzYsInVzZXJfbmFtZSI6InBldGVyliwiYXV0aG9y
aXRpZXMiOlsIuk9MRV9VU0VSII0sImp0aSI6IjkMmJiNjQ4LTQ2MWQtNGVlYy1hZTljLTlYWyUxZjA4ZTjhMilsImNsawVvudF9pZCI
6lmFwcGxpY2FoWaW9uaWQiLCjY29wZSI6WyJmb28iXX0.tr4yUmGLtsH7q9Ge2i7gxyTsOOaORSOYoc2uBuAW5OVIKZcVsIITWV3
bDN0FVBzimpAPy33tvicFRohBFoVThqKxzG00SkURN5bnQ4uFLAP0NjZ6BuDjvMwvXNxRQp2lVXl4lQ4eTvuyZozjUSCXzC1LN
w5EFFi22j73g1_mRm2j-dEhBp1TvMaRKLBdk2hzlDVKzu5oj_gODBFm3a1-
IijYoCimIm2igcesXkipRjtjNcrjSegBbGgyXHVak2gB7l07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97jIT6Rr1XZBlveyjfKs_
XIXgU5qizRm9mt5xg","token_type":"bearer","refresh_token":"","expires_in":5999,"scope":"foo","jti":"4d2bb648-461d-4eec-
ae9c-5cae1f08e2a2"}
```

Now let's invoke the Order Processing microservice with the JWT you got from the previous `curl` command. Set the same JWT you got in the HTTP Authorization Bearer header using the following `curl` command and invoke the Order Processing microservice. Because the JWT is a little lengthy, you can use a small trick when using the `curl` command. Export the JWT to an environmental variable (`TOKEN`), then use that environmental variable in your request to the Order Processing microservice.

```
\> export TOKEN=jwt_access_token
\> curl -v -k -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" -d
  '{"customer_id": "101021", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John
  Doe", "billing_address": "201, 1st Street, San Jose,
  CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}'
https://35.247.11.161/orders
```

In the previous command, we do an HTTP POST to the Order Processing microservice so that the Order Processing microservice can talk to the Inventory microservice. In return, you won't get any JSON payload, but only an HTTP 201 status code. When the Order Processing microservice talks to Inventory microservice, the Inventory microservice prints the item codes

in its logs. You can tail the logs with the following command that includes the pod name corresponding to the Inventory microservice:

```
\> kubectl logs inventory-deployment-f7b8b99c7-4t56b --follow
```

11.6 Using Kubernetes service accounts

Kubernetes uses two types of accounts for authentication and authorization: user accounts and service accounts. The user accounts aren't created or managed by Kubernetes, while the service accounts are. In this section we discuss how Kubernetes manages service accounts and associates those with pods.

In appendix B we talked about the high-level Kubernetes architecture and how a Kubernetes node communicates with the API server. Kubernetes uses service accounts to authenticate a pod to the API server. A service account provides an identity to a pod, and Kubernetes uses the `ServiceAccount` object to represent a service account. Let's use the following command to list all the service accounts available in our Kubernetes cluster (under the default namespace):

```
\> kubectl get serviceaccounts  
NAME    SECRETS AGE  
default  1        11d
```

By default, at the time you create a Kubernetes cluster, Kubernetes also creates a service account for the default namespace. To find more details about the default service account, use the following `kubectl` command. It lists the service account definition in yaml format. There you can see that the default service account is bound to the default token secret that we discussed in section 11.3.1.

```
\> kubectl get serviceaccount default -o yaml  
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  creationTimestamp: "2019-09-17T02:01:00Z"  
  name: default  
  namespace: default  
  resourceVersion: "279"  
  selfLink: /api/v1/namespaces/default/serviceaccounts/default  
  uid: ff3d13ba-d8ee-11e9-a88f-42010a8a01e4  
secrets:  
- name: default-token-19fj8
```

Kubernetes binds each pod to a service account. There can be multiple pods bound to the same service account, but there can't be multiple service accounts bound to the same pod. For example, the default behavior is the default service account created when creating the Kubernetes a namespace is assigned to all the pods created under the corresponding namespace. Under each namespace you'll find a service account called `default`.

11.6.1 Creating a service account and associating it with a pod

In this section we create a service account called ecomm, and update the STS deployment to use it. In other words, we want all the pods running under the STS deployment to run under the ecomm service account. Let's use the following `kubectl` command to create the ecomm service account:

```
\> kubectl create serviceaccount ecomm  
serviceaccount/ecomm created
```

At the time of creating the service account, Kubernetes also created a token secret and attached it to the service account. When we update the STS deployment to run under the ecomm service account, all the pods under the STS deployment can use this token secret to authenticate to the API server. The following command shows the details of the ecomm service account in yaml format:

```
\> kubectl get serviceaccount ecomm -o yaml  
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: ecomm  
  namespace: default  
secrets:  
- name: ecomm-token-92p7g
```

Now let's set the ecomm service account for the STS deployment. The complete updated definition of the STS deployment is in chapter11/sample01/sts.deployment.with.service.account.yaml file. We introduced these new changes to the STS deployment created in section 11.3.2. As shown in listing 11/17, the only change was to add the `serviceAccountName` element under the `pod spec:` of the deployment.

Listing 11.17. Attaching a service account to the pod

```
spec:  
  serviceAccountName: ecomm  
  containers:  
    - name: sts  
      image: prabath/secure-sts-ch10:v1  
      imagePullPolicy: Always  
    ports:  
      - containerPort: 8443
```

Let's use the following command from the chapter11/sample01 directory to update the deployment:

```
\> kubectl apply -f sts.deployment.with.service.account.yaml  
deployment.apps/sts-deployment configured
```

If you run the `kubectl describe pod` command against the pod Kubernetes created under the STS deployment now, you'll find that it uses the token secret Kubernetes automatically created for the ecomm service account.

11.6.2 Benefits of running a pod under a custom service account

If you don't specify a service account under the pod `spec5` of a deployment (listing 11.17), Kubernetes runs all the corresponding pods under the same default service account, created under the same Kubernetes namespace.

NOTE Having different service accounts for each pod or for a group of pods helps you isolate what each pod can do with the Kubernetes API server.

This is one security best practice we should follow in a Kubernetes deployment. Then again, even if you've different service accounts for different pods, if you don't enforce authorization checks at the API server, it adds no value. GKE enables role-based access control by default.

If your Kubernetes cluster doesn't enforce authorization checks, there's another option. If you don't want your pod to talk to the API server at all, then you can ask Kubernetes not to provision the default token secret to that corresponding pod. Without the token secret, none of pods will be able to talk to the API server. To disable the default token provisioning, you need to set the `automountServiceAccountToken` element to `false` under the pod `spec:` of the deployment (listing 11.17).

11.7 Role-based access control (RBAC) in Kubernetes

Role-based access control (RBAC) in Kubernetes defines what actions a user or a service can perform in a Kubernetes cluster. A *role*, in general, defines a set of permissions or capabilities. Kubernetes has two types of objects to represent a role: `Role` and `ClusterRole`. The `Role` object represents capabilities associated with Kubernetes resources within a namespace, while the `ClusterRole` represents capabilities at the Kubernetes cluster level.

Kubernetes defines two types of bindings to bind a role to one or more users (or service): `RoleBinding` and `ClusterRoleBinding`. The `RoleBinding` object represents a binding of namespaced resources (or a `Role`) to a set of users or services, while the `ClusterRoleBinding` object represents a binding of cluster-level resources (or `ClusterRole`) to a set of users or services. Let's use the following command to list all the `ClusterRoles` available in your Kubernetes environment. The truncated output shows the `ClusterRoles` available in GKE by default.

⁵The pod spec in a Kubernetes deployment object defines the parameters for the corresponding pod. A given Kubernetes deployment can have more than one pod spec.

```
\> kubectl get clusterroles
NAME          AGE
admin         12d
cloud-provider 12d
cluster-admin 12d
edit          12d
gce:beta:kubelet-certificate-bootstrap 12d
gce:beta:kubelet-certificate-rotation 12d
gce:cloud-provider 12d
kubelet-api-admin 12d
system:aggregate-to-admin 12d
system:aggregate-to-edit 12d
system:aggregate-to-view 12d
```

To view the capabilities of a given ClusterRole, let's use the following `kubectl` command. The output in yaml format shows that under the rules section, the `kubelet-api-admin` role can perform any verb (or action) on the `nodes/proxy`, `nodes/log`, `nodes/stats`, `nodes/metrics`, and `nodes/spec` resources.

```
\> kubectl get clusterrole kubelet-api-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
labels:
  addonmanager.kubernetes.io/mode: Reconcile
  kubernetes.io/cluster-service: "true"
name: kubelet-api-admin
rules:
- apiGroups:
  - ""
resources:
- nodes/proxy
- nodes/log
- nodes/stats
- nodes/metrics
- nodes/spec
verbs:
- '*'
```

Let's use the following command to list all the ClusterRoleBindings available in your Kubernetes environment. The truncated output shows the ClusterRoleBindings available in GKE by default.

```
\> kubectl get clusterrolebinding
NAME          AGE
cluster-admin 12d
event-exporter-rb 12d
gce:beta:kubelet-certificate-bootstrap 12d
gce:beta:kubelet-certificate-rotation 12d
gce:cloud-provider 12d
heapster-binding 12d
kube-apiserver-kubelet-api-admin 12d
kubelet-bootstrap 12d
```

To view the users and services attached to a given ClusterRoleBinding, let's use the following `kubectl` command. The output of the command, in yaml, shows that under the `roleRef:` section, `kube-apiserver-kubelet-api-admin` refers to the `kublet-api-admin` ClusterRole, and under the `subjects:` section, the `kube-apiserver` user is part of the role binding. Or, in other words, `apiserver-kubelet-api-admin` binds the `kube-apiserver` user to the `kublet-api-admin` ClusterRole.

```
\> kubectl get clusterrolebinding kube-apiserver-kubelet-api-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
labels:
  addonmanager.kubernetes.io/mode: Reconcile
  kubernetes.io/cluster-service: "true"
name: kube-apiserver-kubelet-api-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kubelet-api-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: kube-apiserver
```

As we discussed in section 11.5, Kubernetes has two types of accounts: users and service accounts, and users aren't managed by Kubernetes. In this case, we've a user (not a service account) and that user is managed by GKE.

11.7.1 Talking to the Kubernetes API server from the STS

Let's say, for example, we need the STS to talk to the API server. Ideally, we'll do that in the STS code itself. Because this is just an example, we'll use cURL from a container that runs the STS. Use the following `kubectl` command to directly access the shell of an STS pod. Because we only have one container in each pod, we can simply use the pod name (`sts-deployment-69b99fc78c-j76t1`) here.

```
\> kubectl -it exec sts-deployment-69b99fc78c-j76t1 sh
#
```

After you run the command, you end up with a shell prompt within the corresponding container. Also, we assume that you've followed along in section 11.5.1 and updated the STS deployment, where now it runs under the ecomm service account.

Because we want to use cURL to talk to the API server, we need to first install cURL with the following command in the STS container. And because the containers are immutable, during this exercise if you restart the pod, you'll need to install cURL again.

```
# apk add --update curl && rm -rf /var/cache/apk/*
```

To invoke an API, we also need to pass the default token secret (which is a JWT) in the HTTP authorization header. Let's use the following command to export the token secret to the `TOKEN` environment variable. As we've previously mentioned, the default token secret is accessible to every container from the `/var/run/secrets/kubernetes.io/serviceaccount/token` file.

```
# export TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

The following `curl` command talks to the Kubernetes API server to list all the metadata associated with the current pod. Here we pass the default token secret, which we exported to the `TOKEN` environment variable in the HTTP authorization header. Also, inside a pod, Kubernetes itself populates the value of `HOSTNAME` environment variable with the corresponding pod name, and the `kubernetes.default.svc` hostname is mapped to the IP address of the API server running in the Kubernetes control plane.

```
# curl -k -v -H "Authorization: Bearer $TOKEN" https://kubernetes.default.svc/api/v1/namespaces/default/pods/$HOSTNAME
```

In response to this command, the API server returns the HTTP 403 code, which means the ecomm service account isn't authorized to access this particular API. In fact, it's not only this specific API, the ecomm service account isn't authorized to access any of the APIs on the API server! That's the default behavior of GKE. Either the default service account that Kubernetes creates for each namespace or a custom service account you create aren't associated with any roles.

11.7.2 **Associating a service account with a ClusterRole**

There are two ways to associate the ecomm service account with a ClusterRole. By associating this account as required, we can repeat the exercise in section 11.6.1 and get a successful response. Associating a service account with a ClusterRole gives that particular account the permissions to do certain tasks authorized under the corresponding ClusterRole.

One way to associate a service account with a ClusterRole is to create a ClusterRoleBinding; the other way is to update an existing ClusterRoleBinding. In this section, we follow the latter and update the existing `apiserver-kubelet-api-admin` ClusterRoleBinding in GKE. You can find the updated definition of the `apiserver-kubelet-api-admin` ClusterRoleBinding in the `chapter11/sample01/ecomm.clusterrole.binding.yaml` file (and also in listing 11.18).

Listing 11.18. The definition of kubelet-apiserver-kubelet-api-admin

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    kubernetes.io/cluster-service: "true"
  name: kube-apiserver-kubelet-api-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kubelet-api-admin
```

```
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: kube-apiserver
- kind: ServiceAccount
  namespace: default
  name: ecomm
```

Let's use the following command from the chapter11/sample01 directory to update the deployment with the updated ClusterRoleBinding:

```
\> kubectl apply -f ecomm.clusterrole.binding.yaml
clusterrolebinding.rbac.authorization.k8s.io/kube-apiserver-kubelet-api-admin configured
```

Now if you redo the exercise in section 11.6.1, you'll get a successful response, as the ecomm service account is authorized to perform that action on the API server.

11.8 Summary

- Kubernetes uses ConfigMaps to externalize configurations from the application code, which runs in a container, but it's not the correct way of externalizing sensitive data in Kubernetes.
- The ideal way to store sensitive data in a Kubernetes environment is to use Secrets, where Kubernetes stores the value of a Secret in its etcd distributed name/value pair in an encrypted format.
- Kubernetes dispatches Secrets only to the pods that use them, and even in such cases, the Secrets are never written to disk, only kept in memory.
- Each pod, by default, is mounted with a token secret, which is a JSON Web Token (JWT). A pod can use this default token secret to talk to the Kubernetes API server.
- Kubernetes has two types of accounts: users and service accounts. The user accounts aren't created or managed by Kubernetes, while the service accounts are.
- By default, each pod is associated with a service account (with the name, default), and each service account has its own token secret.
- It's recommended that you always have different service accounts for different pods. This is one of the security best practices we should always follow in a Kubernetes deployment.
- If you have a pod that doesn't need to access the API server, it's recommended that you not provision the token secret to such pods.
- Kubernetes uses Roles/ClusterRoles and RoleBindings/ClusterRoleBindings to enforce access control on the API server.

12

Securing microservices with Istio service mesh

This chapter covers

- Terminating Transport Layer Security (TLS) at the Istio ingress gateway
- Securing service-to-service communications with mutual Transport Layer Security (mTLS) in an Istio environment
- Securing service-to-service communication with JSON Web Token (JWT) in an Istio environment
- Enforcing Role-base Access Control (RBAC) with Istio
- Managing keys in an Istio deployment

In chapter 6 we discussed how to secure service-to-service communication with certificates and in chapter 7 we extended that discussion to use JSON Web Tokens (JWT) to secure service-to-service communication. Then in chapters 10 and 11 we discussed how to deploy a set of microservices as Docker containers in Kubernetes and then again secure service-to-service communication with JWT over mutual Transport Layer Security (mTLS). In all of these cases each microservice by itself had to worry about doing security processing. Or in other words, each microservice embedded a set of Spring Boot libraries to do security processing. This violates one key aspect of microservices architecture, the Single Responsibility Principle,¹ under which a microservice should be performing only one particular function.

A Service Mesh is a result of the natural evolution in implementing the Single Responsibility Principle in microservices architecture. It is in fact an architectural pattern that brings in the

¹ https://en.wikipedia.org/wiki/Single_responsibility_principle

best practices in resiliency, security, observability, and routing control to your microservices deployment, which we discuss in appendix C in detail. Istio is an *out-of-process Service Mesh* implementation that runs apart from your microservice, transparently intercepts all the traffic coming into and going out from your microservice; and enforces security controls, and so on.

NOTE If you are new to the Service Mesh pattern and Istio please go through appendix C first. This chapter assumes you understand the Service Mesh pattern and Istio, and have gone through all the samples discussed in the appendix C.

In this chapter we are going to redo some of the examples we discussed in chapters 6 and chapter 7, following the Service Mesh architecture with Istio.

12.1 Setting up the Kubernetes deployment

In this section we are going to setup a Kubernetes deployment (figure 12.1), which is very much similar to what we had in chapter 11 under section 11.5. The only difference is, here we have removed the JWT verification and mutual TLS from the Order Processing microservice and Inventory microservice. Also we have removed TLS support from all three services (figure 12.1).

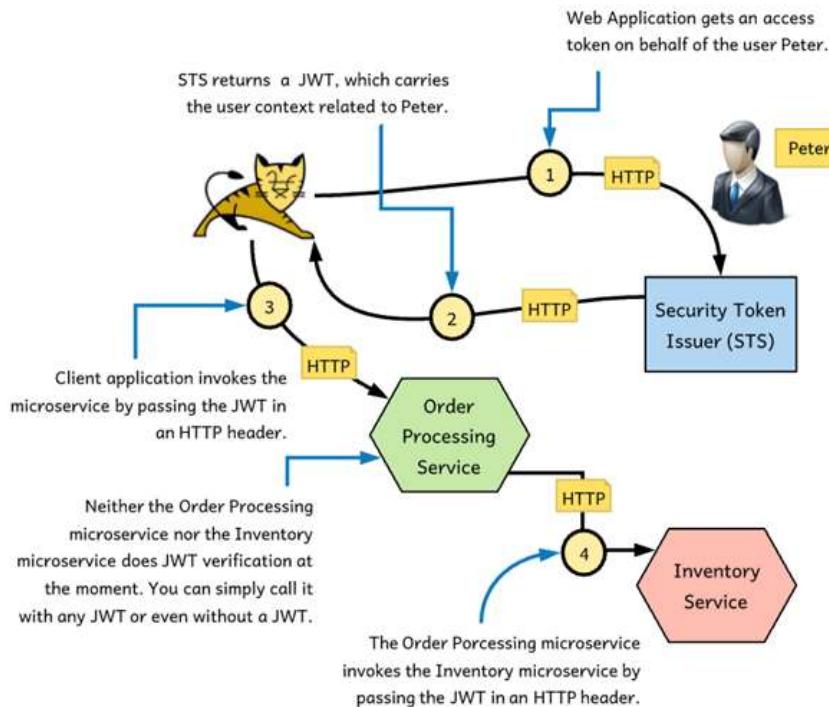


Figure 12.1 STS issues a JWT access token to the web application and the web application uses it to access the microservice, on behalf of the user, Peter. The Order Processing microservice uses the same JWT it got from the client application to access the Inventory microservice.

12.1.1 Enabling Istio auto injection

To add Istio support for the microservices we are about to deploy, or in other words to auto inject Envoy as a sidecar proxy to the microservice deployment (see figure 12.2), run the following `kubectl` command. Here we enable auto injection for the `default` namespace. In appendix C, we discussed how to install Istio in Kubernetes and sidecar auto injection in detail, under section C.11.1.

```
\> kubectl label namespace default istio-injection=enabled
```

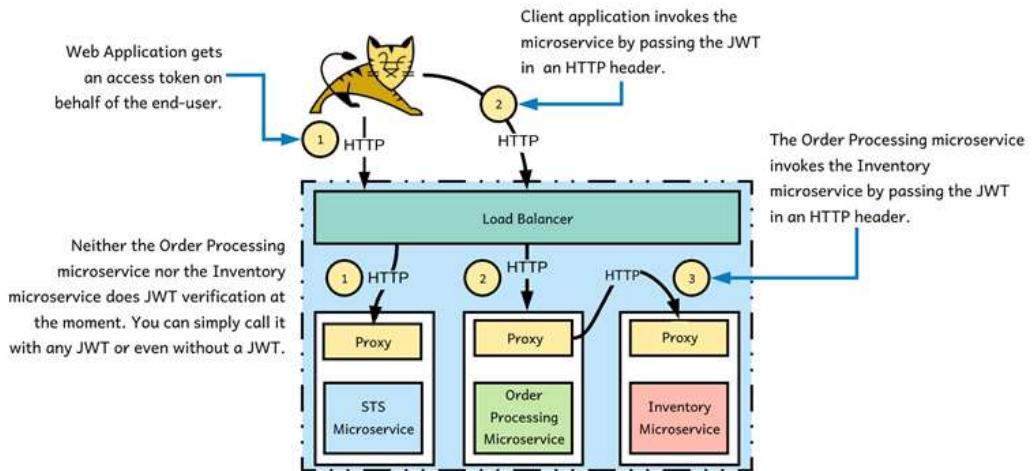


Figure 12.2 Istio will introduce the Envoy sidecar proxy to each pod, along with the container that carries the microservice.

12.1.2 Clean up any previous work

If you have already run the samples in appendix-c or chapter 11, let's run the following command from `chapter12/sample01` directory to clean those deployments. All the source code related to the samples in this chapter is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, under the `chapter12` directory. The `clean.sh` script deletes all the Kubernetes Services, Deployments, ConfigMaps and Secrets we created in appendix C and chapter 11.

```
\> sh clean.sh
```

12.1.3 Deploying microservices

To create three Kubernetes Deployments and the corresponding Services, run the following command from `chapter12/sample01` directory.

```
\> kubectl apply -f .
configmap/inventory-application-properties-config-map created
deployment.apps/inventory-deployment created
service/inventory-service created
configmap/orders-application-properties-config-map created
deployment.apps/orders-deployment created
service/orders-service created
configmap/sts-application-properties-config-map created
configmap/sts-jwt-keystore-config-map created
secret/sts-keystore-secrets created
deployment.apps/sts-deployment created
service/sts-service created
```

Let's run the following command to verify the deployment. This shows three pods, each corresponding to each microservice. In each pod we see two containers, one for the microservice and the other for the Envoy proxy.

| \> kubectl get pods | | | | |
|---------------------------------------|-------|---------|----------|--|
| NAME | READY | STATUS | RESTARTS | |
| inventory-deployment-7848664f49-x9z7h | 2/2 | Running | 0 | |
| orders-deployment-7f5564c8d4-ttgb2 | 2/2 | Running | 0 | |
| sts-deployment-85cdd8c7cd-qk2c5 | 2/2 | Running | 0 | |

Next, you can run the following command to list the available Services. Here we have `order-service` and `sts-service` of LoadBalancer type². That means these two services are publicly accessible. When we use Istio, we do not need to make these services of LoadBalancer type, because we anyway going to use the Istio ingress gateway – and all the traffic should flow through it. If we have a Service of LoadBalancer type, and also expose that via the Istio ingress gateway, then we create two entry points for that microservice. One can access it with the external IP address of the Service, and also with the IP address of the Istio ingress gateway, which does not sound good.

| \> kubectl get services | | | | |
|-------------------------|--------------|---------------|----------------|--------------|
| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) |
| inventory-service | ClusterIP | 10.39.243.179 | <none> | 80/TCP |
| orders-service | LoadBalancer | 10.39.242.8 | 35.247.11.161 | 80:32515/TCP |
| sts-service | LoadBalancer | 10.39.243.231 | 35.199.169.214 | 80:31569/TCP |

If you look at the PORT column in the above output, you would notice that all three Services are running on HTTP on port 80 (not over TLS), which is not good in terms of security. In section 12.2, we discuss how to use Istio ingress gateway to protect these Services with TLS. Also – in section 12.2, we are going to redeploy both the `order-service` and `sts-service` Services as ClusterIP Services – so they won't be directly accessible outside the Kubernetes cluster.

12.1.4 Testing end-to-end flow

In this section, we are going to test the end-to-end flow as shown in figure 12.1. Since the Order Processing microservice is secured with JWT, first we need to get a JWT from the STS, which the Order Processing microservice trusts. Let's run the following cURL command to talk to the STS and get a JWT. Here we use cURL to represent your client application:

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" http://35.199.169.214/oauth/token #A
```

#A Make sure to use the IP address corresponding to your security token service (STS).

²If you are not familiar with different Kubernetes Service types please check appendix B.

The following command does an HTTP GET to the Order Processing microservice. Here we do not need to pass the JWT we obtained from the previous command, because we are not doing JWT verification at the service level. Later in the chapter we discuss how to do JWT verification with Istio.

```
\> curl -k http://35.247.11.161/orders/11 #A
```

#A Make sure to use the IP address corresponding to your Order Processing microservice.

Finally let's run the following command to do a POST to the Order Processing microservice, which internally calls Inventory microservice:

```
\> curl -v -k -H "Content-Type: application/json" -d
  '{"customer_id":"101021","payment_method":{"card_type":"VISA","expiration":"01/22","name":"John Doe","billing_address":"201, 1st Street, San Jose, CA"}, "items":[{"code":"101","qty":1}, {"code":"103","qty":5}], "shipping_address":"201, 1st Street, San Jose, CA"}' http://35.247.11.161/orders #A
```

#A Make sure to use the IP address corresponding to your Order Processing microservice.

All the communications here happen over HTTP. This is only an incomplete setup to verify that our Kubernetes deployment works fine and in an ideal production deployment we should not expose any of the microservices just over HTTP. As we move forward in this chapter, we discuss how to make this incomplete setup more complete and secure.

12.2 Enabling TLS termination at the Istio ingress gateway

In this section we are going to expose the three microservices we deployed in Kubernetes in section 12.1 via the Istio ingress gateway over TLS. The Istio ingress gateway runs at the edge of the Kubernetes cluster or in other words, all the traffic coming to your service mesh, will first go through the ingress gateway (figure 12.3). Here we only enable TLS at the ingress gateway – and the ingress gateway terminates TLS at the edge and then routes traffic over HTTP to the corresponding microservices. In section 12.3 we discuss how to enable mutual TLS between ingress gateway and the microservices – and also between microservices.

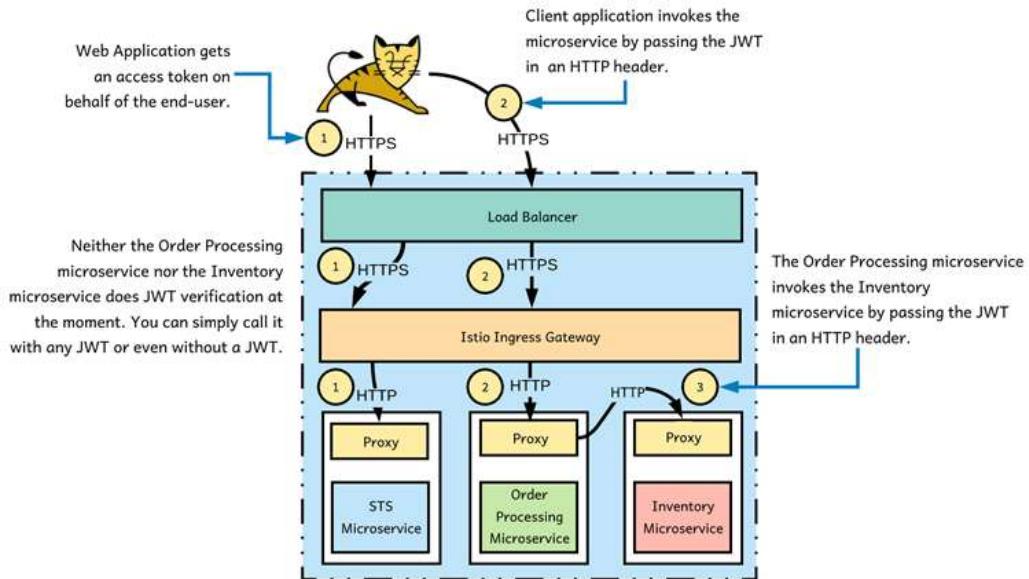


Figure 12.3 Istio ingress gateway intercepts all the requests coming to the microservice and terminates the TLS connection.

12.2.1 Deploying TLS certificates to the Istio ingress gateway

To enable TLS at the ingress gateway, first we need to create a public/private key pair. Here we use OpenSSL to generate the keys. OpenSSL is a commercial-grade toolkit and cryptographic library for Transport Layer Security (TLS), available for multiple platforms. We discuss in appendix K, how to generate keys with OpenSSL in detail. The following command uses an OpenSSL Docker image, which we can use to generate keys, so you need not to worry about installing OpenSSL locally. Run the following command from `chapter12/sample02` directory and once the OpenSSL container spins up, it provides a prompt to execute OpenSSL command.

```
\$ docker run -it -v $(pwd):/export prabath/openssl
#
```

Run the following OpenSSL command to generate a public/private key pair – and you can find them inside the `chapter12/sample02/gateway-keys` directory. When you pull the samples from the GitHub repository, the keys are already in the `chapter12/sample02/gateway-keys` directory – so, before executing the following command make sure to delete them. Or else, you can skip the key generation and use the keys from the GitHub repository.

```
# openssl req -nodes -new -x509 -keyout /export/gateway-keys/server.key -out
/export/gateway-keys/server.cert -subj "/CN=gateway.ecomm.com"
```

The Istio ingress gateway reads the public/private key pair for the TLS communication from a well-defined Kubernetes Secret called, `istio-ingressgateway-certs`. In fact Istio ingress gateway is an Envoy proxy, running within a pod, under the `istio-system` namespace, in the Kubernetes cluster. You can run the following command to list all the pods available in the `istio-system` namespace and find the name of the istio ingress gateway pod. The output of the command is truncated only to show the istio ingress gateway pod.

```
\> kubectl get pods -n istio-system
NAME                               READY   STATUS    RESTARTS
istio-ingressgateway-6d8f9d87f8-sc2ch   1/1     Running   0
```

To learn more about the Istio ingress gateway pod let's use the following command in listing 12.1 with the correct pod name. The listing 12.1 only shows a truncated output with the volume mounts (which we discussed in appendix B and chapter 11) and the code annotations explain some important elements.

Listing 12.1. The volume mounts of the istio ingress gateway pod

```
\> kubectl get pod -n istio-system istio-ingressgateway-6d8f9d87f8-sc2ch -o yaml
volumeMounts:
- mountPath: /etc/certs      #A
  name: istio-certs
  readOnly: true
- mountPath: /etc/istio/ingressgateway-certs    #B
  name: ingressgateway-certs
  readOnly: true
- mountPath: /etc/istio/ingressgateway-ca-certs  #C
  name: ingressgateway-ca-certs
  readOnly: true
- mountPath: /var/run/secrets/kubernetes.io/serviceaccount    #D
  name: istio-ingressgateway-service-account-qpmwf
  readOnly: true
```

#A Mount path from the Envoy local file system for the keys used by Envoy when talking to upstream services protected with mTLS, identified by the volume name `istio-certs`.

#B Mount path from the Envoy local file system for the private/public key pair to enable TLS for the downstream clients.

#C Envoy used the public certificates of trusted certificate authorities to authenticate downstream clients when using mTLS.

#D The default JWT, which is provisioned by Kubernetes – not specific to Istio.

To create the `istio-ingressgateway-certs` from the private/public keys we created, lets run the following command from `chapter12/sample02` directory.

```
\> kubectl create secret tls istio-ingressgateway-certs --key gateway-keys/server.key --cert
      gateway-keys/server.cert -n istio-system
secret/istio-ingressgateway-certs created
```

Finally, to enable TLS at the ingress gateway, we need to define a Gateway resource. The Gateway resource, introduced by Istio, instructs the load balancer on how to route traffic to the Istio ingress gateway. You can find more details on the Gateway resource in appendix C. Since the ingress gateway is running in the `istio-system` namespace, we also create the Gateway

resource in the same namespace. Ideally you only need to have one Gateway resource for your Kubernetes cluster. That is only a common practice. Another practice is to have a Gateway resource in every namespace, but all of them pointing to the same Istio ingress gateway pod running in the `istio-system` namespace. The listing 12.2 shows the definition of the Gateway resource. Here the `ecomm-gateway` Gateway resource instructs the load balancer to route traffic on HTTPS port 443 and HTTP port 80 to the Istio ingress gateway pod, which is an Envoy proxy.

Listing 12.2. The definition of the Gateway resource

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ecomm-gateway
  namespace: istio-system    #A
spec:
  selector:
    istio: ingressgateway     #B
  servers:
    - port:      #C
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE      #D
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt    #E
      privateKey: /etc/istio/ingressgateway-certs/tls.key    #F
    hosts:
      - "*"        #G
    - port:      #H
      number: 80
      name: http
      protocol: HTTP
    hosts:
      - "*"
  tls:
    httpsRedirect: true    #I
```

#A The Gateway resource is created in `istio-system` namespace

#B Associates the `istio` ingress gateway pod with the Gateway resource

#C Instructs the load balancer route traffic on port 443 on HTTPS to the ingress gateway

#D Enables TLS

#E The volume mount of the public certificate for TLS communication

#F The volume mount of the private key for the TLS communication

#G Instructs the load balancer route traffic on port 443 on HTTPS to the ingress gateway for any host name

H Instructs the load balancer to route traffic on port 80 on HTTP to the ingress gateway for any host name

#I Redirects any HTTP traffic to the HTTPS endpoint

In listing 12.2 we use SIMPLE as the `tls/mode`. In addition to that Istio supports four more modes, as follows:

- SIMPLE: Enables one-way TLS communication between the client applications and the ingress gateway. You need to have a public/private key pair for the gateway.

- PASSTHROUGH: Just passes through the traffic to the corresponding pod, based on the Server Name Indication (SNI³). In appendix C, under section C.13 we discuss a sample, which uses PASSTHROUGH mode. With this mode, you do not need to configure any public/private key pair for the gateway.
- MUTUAL: Enables mutual TLS communication between the client applications (downstream applications) and the ingress gateway. You need to have a public/private key pair for the gateway and also should have a set of certificate authority (CA) certificates. Only client applications carrying a valid certificate signed by any of the trusted CAs can access the ingress gateway. To enable mutual TLS, we need to create a Secret with the name, `istio-ingressgateway-ca-certs` which carries CA certificates and define a new element called `caCertificates` under the `tls` element of the Gateway resource, which carries the value `/etc/istio/ingressgateway-ca-certs/ca-chain.cert.pem`.
- AUTO_PASSTHROUGH: Under the PASSTHROUGH mode, the ingress gateway expects an Istio VirtualService to present and reads the configuration data from the corresponding VirtualService to route the traffic to the upstream pod. Under AUTO_PASSTHROUGH mode, we do not need to have a VirtualService, and the ingress gateway expects all the required routing information to be encoded into the SNI value.
- ISTIO_MUTUAL: Just like MUTUAL mode, this too enables mutual TLS communication between the client applications (downstream applications) and the ingress gateway. Unlike in MUTUAL mode, this mode uses the certificates provisioned by Istio itself, which are available under `/etc/certs` location of the Envoy file system.

To deploy the Gateway resource shown in listing 12.2 under the `istio-system` namespace, run the following command from `chapter12/sample02` directory:

```
\> kubectl apply -f istio.public.gateway.yaml
gateway.networking.istio.io/ecomm-gateway created
```

12.2.2 Deploying VirtualServices

In this section we are going to define and deploy two Istio VirtualServices for the STS and the Order Processing microservices. These are two services we want the client applications to access through the Istio ingress gateway. Since the Inventory microservice is only invoked by the Order Processing microservice within the cluster, we do not need to create a VirtualService for it. Before creating the VirtualServices, we need to update the Kubernetes Service definition of the STS and the Order Processing microservices, which we created in section 12.1. There we used the LoadBalancer service type – because we wanted to test the end-to-end flow from the cURL client outside the Kubernetes cluster. Anyway when we expose these microservices through the ingress gateway we don't want the client applications to access them also via the Kubernetes Service directly.

³ Server Name Indication (SNI) is a TLS extension, a client application can use before the start of the TLS handshake to indicate the server, which host name it intends to talk to. Istio gateway can route traffic looking at this SNI parameter.

Run the commands in listing 12.3 from chapter12/sample02 directory to update the two microservices to run as a ClusterIP service (this is the default Service type in Kubernetes, when you have no type defined). First, we delete the two services and then create them.

Listing 12.3. Update the Service definition of STS and Order Processing microservice

```
\> kubectl delete service sts-service    #A
service "sts-service" deleted

\> kubectl delete service orders-service   #B
service "orders-service" deleted

\> kubectl apply -f order.processing.yaml   #C
configmap/orders-application-properties-config-map unchanged
deployment.apps/orders-deployment unchanged
service/orders-service created

\> kubectl apply -f sts.yaml    #D
configmap/sts-application-properties-config-map unchanged
configmap/sts-jwt-keystore-config-map unchanged
secret/sts-keystore-secrets configured
deployment.apps/sts-deployment unchanged
service/sts-service created

#A Deletes the Service definition of the Order Processing microservice
#B Deletes the Service definition of the STS microservice
#C Creates the Service definition of the Order Processing microservice
#D Creates the Service definition of the STS microservice
```

Now we can create two VirtualServices, one for the STS and the other one for the Order Processing microservice. These two virtual services read routing information from the corresponding Kubernetes Services. The listing 12.4 shows the VirtualService definition corresponding to the Order Processing microservice.

Listing 12.4. The VirtualService definition of Order Processing microservice

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: orders-virtual-service
spec:
  hosts:
  - orders.ecomm.com    #A
  gateways:
  - ecomm-gateway.istio-system.svc.cluster.local    #B
  http:
  - route:    #C
    - destination:
        host: orders-service
        port:
          number: 80
```

#A This VirtualService only applicable for requests coming to the orders.ecomm.com host name
#B The full qualified name of the Gateway resource created in istio-system namespace

#C Route the traffic to the order-service Service on port 80. Istio finds the Cluster IP of the order-service Service and routes the traffic there

The VirtualService definition corresponding to the STS too looks similar to the Order Processing microservice's definition, which you can find in the chapter12/sample02/virtualservices.yaml file. Something very important to highlight here is, for Istio traffic rules to work, we must use named ports (listing 12.5) in the Kubernetes Service definition corresponding to the destination routes in the VirtualService. Also, these named ports should carry predefined values as explained in <https://istio.io/docs/setup/additional-setup/requirements/>.

Listing 12.5. The Service definition of Order Processing microservice

```
apiVersion: v1
  kind: Service
  metadata:
    name: orders-service
  spec:
    selector:
      app: orders
    ports:
      - name: http    #A
        protocol: TCP
    port: 80
    targetPort: 8443
```

#A This must be a named port for Istio routing rules to work

To deploy the VirtualServices under the default namespace run the following command from chapter12/sample02 directory.

```
\> kubectl apply -f virtualservices.yaml
virtualservice.networking.istio.io/sts-virtual-service created
virtualservice.networking.istio.io/orders-virtual-service created
```

12.2.3 Defining a permissive authentication policy

In this section we are going to define a policy to instruct Istio not to use mTLS between microservices or between the Istio ingress gateway and the microservices. In other words, we want the ingress gateway to terminate the TLS connection. The policy in listing 12.6 uses PERMISSIVE as the mtlsMode, so Istio won't enforce mTLS or TLS for any requests to the Services defined under the spec/target element.

Listing 12.6. The permissive authentication policy

```
apiVersion: authentication.istio.io/v1alpha1
  kind: Policy
  metadata:
    name: ecomm-authn-policy
  spec:
    targets:    #A
      - name: orders-service    #B
```

```

- name: inventory-service
peers:
- mtls:
  mode: PERMISSIVE #C

```

#A This policy is only applicable to these targets
#B Must match the name of a Kubernetes Service name
#C Instructs Istio to accept plaintext traffic as well as mutual TLS traffic at the same time

Run the following command from `chapter12/sample02` directory to apply the authentication policy defined in listing 12.6.

```
\> kubectl apply -f authentication.policy.yaml
policy.authentication.istio.io/ecomm-authn-policy created
```

12.2.4 Testing end-to-end flow

In this section, we are going to test the end-to-end flow, as shown in figure 12.4, with the Istio ingress gateway secured with TLS. All the communications between the cURL client and the microservices now happen via the ingress gateway. Let's run the following two commands to find the external IP address and the HTTPS port of the Istio ingress gateway, which runs under the `istio-system` namespace. The first command finds the external IP address of the `istio-ingressgateway` service and exports it to the `INGRESS_HOST` environment variable and the second command finds the HTTPS port of the `istio-ingressgateway` service and exports it to the `INGRESS_HTTPS_PORT` environment variable.

```
\> export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o
  jsonpath='{.status.loadBalancer.ingress[0].ip}')

\> export INGRESS_HTTPS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o
  jsonpath='{.spec.ports[?(@.name=="https")].port}')
```

You can use the following echo command to make sure that we captured the right values for the two environment variables.

```
\> echo $INGRESS_HOST
34.83.117.171
\> echo $INGRESS_HTTPS_PORT
443
```

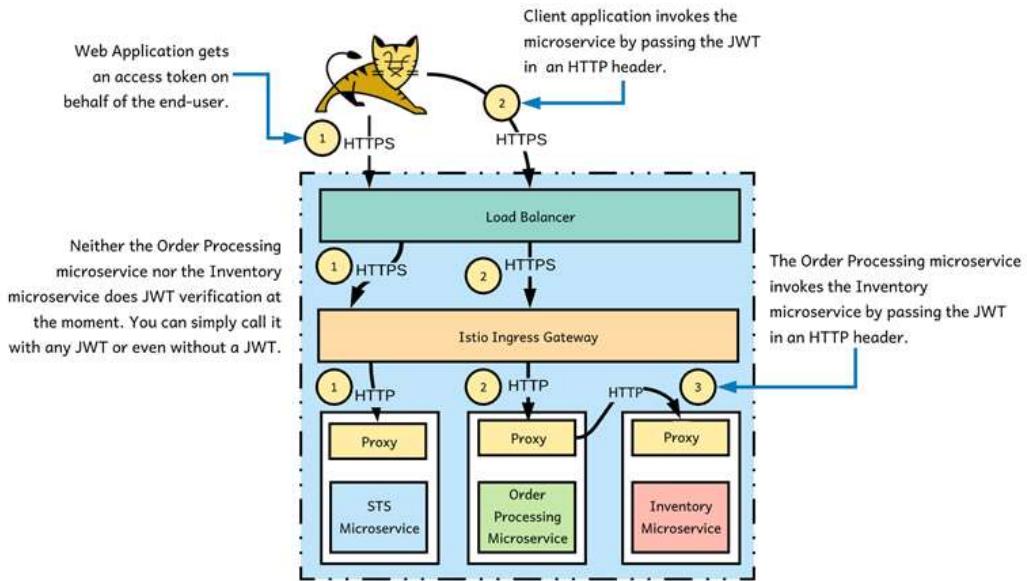


Figure 12.4 STS issues a JWT access token to the web application and the web application uses it to access the microservice, on behalf of the user, Peter. The Order Processing microservice uses the same JWT it got from the client application to access the Inventory microservice. Istio ingress gateway intercepts all the requests coming to the microservice and terminates the TLS connection.

Let's use the following cURL command to talk to the STS and get a JWT. Here we use the environment variables, which we defined before for the hostname and the port of the `istio-ingressgateway` service. Since we are using host name based routing at the Istio gateway and there is no DNS mapping to the hostnames `sts.ecomm.com` or `orders.ecomm.com`, we are using the `--resolve` parameter in cURL to define the hostname to IP mapping.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" --resolve sts.ecomm.com:$INGRESS_HTTPS_PORT:$INGRESS_HOST https://sts.ecomm.com:$INGRESS_HTTPS_PORT/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works fine, the security token service returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token":"eyJhbGciOiJSUzI1NiIs...","token_type":"bearer","refresh_token":"","expires_in":5999,"scope":"foo","jti":"4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

The following command does an HTTP GET to the Order Processing microservice. Before talking to the microservice, let's export the JWT we got from the previous command (under the value

of the `access_token` parameter) to an environmental variable (`TOKEN`). Then use that environmental variable in your request to the Order Processing microservice to carry the JWT along with the HTTP request. Even though we pass the JWT, we are still not doing any JWT verification at the service level. Later in the chapter we discuss how to do JWT verification with Istio.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" --resolve
    orders.ecomm.com:$INGRESS_HTTPS_PORT:$INGRESS_HOST
    https://orders.ecomm.com:$INGRESS_HTTPS_PORT/orders/11
```

Finally let's do a POST to Order Processing microservice, which internally calls the Inventory microservice.

```
\> curl -v -k -H "Authorization: Bearer $TOKEN" --resolve
    orders.ecomm.com:$INGRESS_HTTPS_PORT:$INGRESS_HOST -H "Content-Type: application/json"
    -d
    '{"customer_id": "101021", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}' https://orders.ecomm.com:$INGRESS_HTTPS_PORT/orders
```

TROUBLESHOOTING If you get any errors for the above commands, see appendix C, section 13.4 for troubleshooting tips.

12.3 Securing service-to-service communication with mTLS

This section extends the use case we discussed in section 12.2 by enforcing mutual mTLS between the Istio ingress gateway and the microservices, and also between microservices (figure 12.5). We assume you have successfully completed all the samples we discussed in section 12.2.

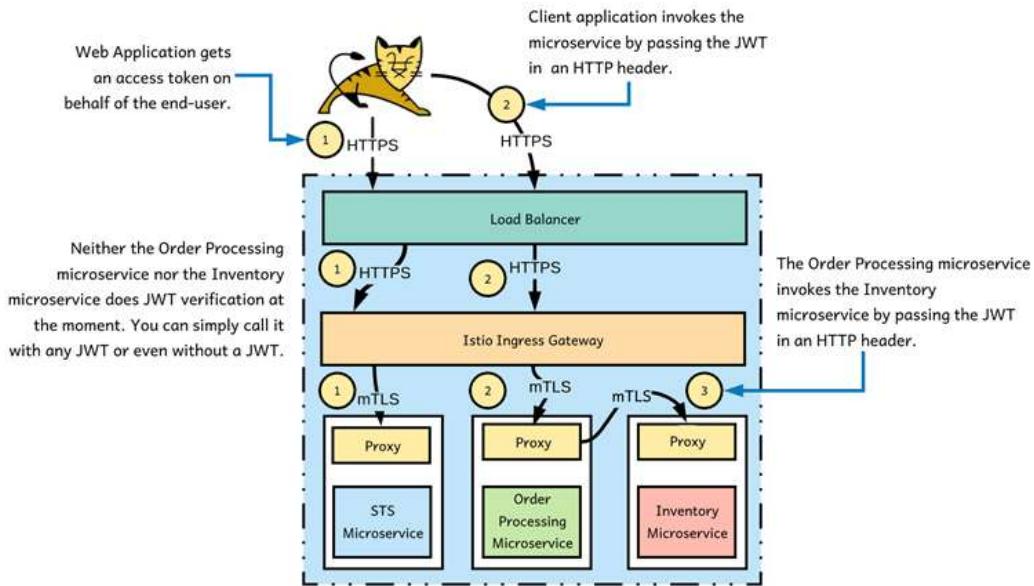


Figure 12.5 Istio ingress gateway intercepts all the requests coming to the microservice and terminates the TLS connection. The communication between the ingress gateway and microservices, and also between microservices, is protected with mTLS.

To enforce mTLS between microservices in a Istio deployment, we need to worry about two ends: the client and the server. Both the mTLS client and server are microservices. In our case, the Istio ingress gateway is a client to the Order Processing microservice, and the Order Processing microservice is a client to the Inventory microservice. In the same way the Order Processing microservice is a server to the ingress gateway and the Inventory microservice is a server to the Order Processing microservice. To enforce mTLS at the server end, we need to update the authentication policy we had in listing 2.6 (which uses `PERMISSIVE`) as the `tls/mode` to use `STRICT` as the `tls/mode` (listing 2.7).

Listing 12.7. The strict mTLS authentication policy

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: ecomm-authn-policy
spec:
  targets:
  - name: orders-service
  - name: inventory-service
  peers:
  - mtls:
    mode: STRICT      #A
```

#A Instructs Istio to enforce mTLS for any traffic comes to the Kubernetes Services defined under targets element

To use mTLS at the client side, when one microservice talks to another microservice, Istio introduces a new resource type called, DestinationRule. The DestinationRule we have in listing 12.8 enforces mTLS for all the communications with all the hosts (indicated by the Kubernetes Service name) in the default namespace

Listing 12.8. The DestinationRule to enforce mTLS between all the Services

```
apiVersion: networking.istio.io/v1alpha3
  kind: DestinationRule
  metadata:
    name: ecomm-authn-service-mtls
  spec:
    host: "*.default.svc.cluster.local"      #A
    trafficPolicy:
      tls:
        mode: ISTIO_MUTUAL      #B
```

#A Specifies the hosts, using Kubernetes Service names

#C Enables mutual TLS communication using the certificates provisioned to each Envoy proxy by Istio itself, which are available under /etc/certs location of the Envoy file system.

To use mTLS at the client side when the Istio ingress gateway talks to a microservice in the default Kubernetes namespace, we use the DestinationRule defined in listing 12.8. We deploy this DestinationRule in the istio-system namespace.

Listing 12.9. The DestinationRule enforcing mTLS between Gateway and Services

```
apiVersion: networking.istio.io/v1alpha3
  kind: DestinationRule
  metadata:
    name: ecomm-authn-istio-gateway-mtls
    namespace: istio-system      #A
  spec:
    host: "*.default.svc.cluster.local"      #B
    trafficPolicy:
      tls:
        mode: ISTIO_MUTUAL      #C
```

#A This DestinationRule is deployed under the istio-system namespace.

#B Specifies the hosts, using Kubernetes Service names

#C Enables mutual TLS communication using the certificates provisioned to each Envoy proxy by Istio itself, which are available under /etc/certs location of the Envoy file system.

Lets run the following command from chapter12/sample03 directory to apply the DestinationRules defined in listing 12.8 and 12.9 and also the authentication policy defined in listing 12.7.

```
\> kubectl apply -f authentication.policy.yaml
policy.authentication.istio.io/ecomm-authn-policy configured
destinationrule.networking.istio.io/ecomm-authn-istio-gateway-mtls created
destinationrule.networking.istio.io/ecomm-authn-service-mtls created
```

To test the end-to-end flow after enforcing mTLS, please follow the steps defined in section 12.2.4.

12.4 Securing service-to-service communication with JWT

This section extends the use case we discussed in section 12.3 by enforcing JWT verification at the each microservice. We assume you have successfully completed all the samples we discussed in section 12.3. Here we use JWT to carry the end-user context, while using mTLS for service-to-service authentication (figure 12.6).

12.4.1 Enforcing JWT authentication

The authentication policy we have in listing 12.7 enforces only mutual TLS between microservices (to be precise, between Envoy proxies), so we need to update it to enforce JWT authentication. The listing 12.10 shows the updated authentication policy and it is applicable for any requests coming to the Order Processing and Inventory microservices.

Listing 12.10. The JWT + mTLS authentication policy

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: ecomm-authn-policy
spec:
  targets: #A
  - name: orders-service
  - name: inventory-service
  peers:
  - mTLS:
      mode: STRICT
  origins:
  - jwt:
      issuer: "sts.ecomm.com" #B
      audiences:
      - "*.ecomm.com" #C
      jwksUri: https://raw.githubusercontent.com/microservices-security-in-
action/samples/master/chapter12/sample04/jwtkey.jwk #D
      principalBinding: USE_ORIGIN #E
```

#A This policy is only applicable to these targets.

#B The value of iss attribute in the JWT (or the issuer of JWT), must exactly match this value

#C The value of the aud attribute in the JWT, must exactly, match this value.

#D The URL to fetch the JSON Web Key, corresponding to the signature of the JWT

#E Set the authenticated principle from origin authentication or the subject of the JWT

Lets run the following command from chapter12/sample04 directory to update the authentication policy authentication policy as defined in listing 12.10.

```
\> kubectl apply -f authentication.policy.yaml
policy.authentication.istio.io/ecomm-authn-policy configured
```

To test the end-to-end flow after enforcing mTLS, please follow the steps defined in section 12.2.4. When you follow the steps in section 12.2.4, unlike in the previous case, if you send an invalid JWT, Istio will reject the request.

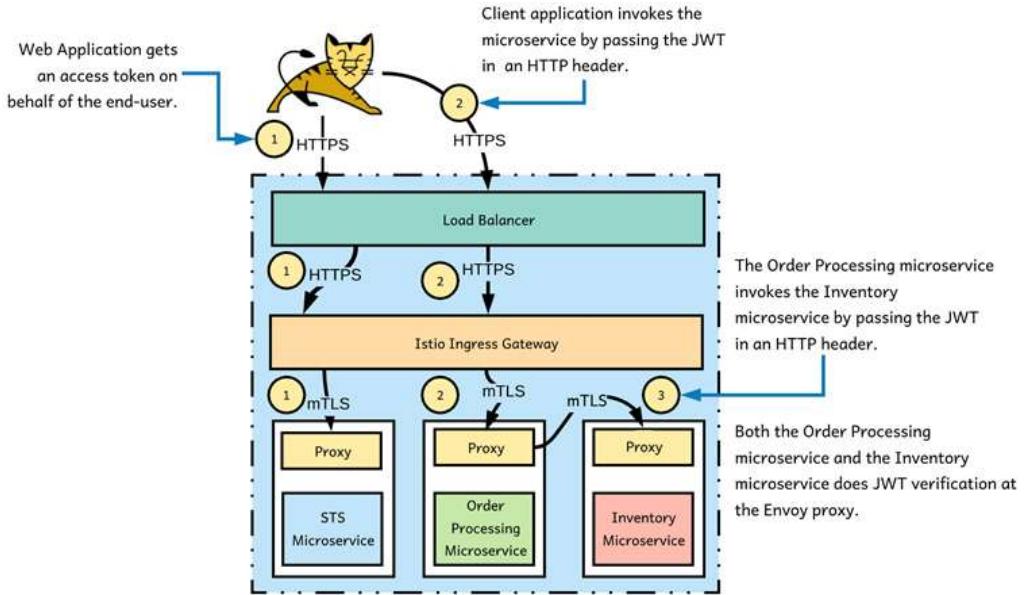


Figure 12.6 Istio ingress gateway intercepts all the requests coming to the microservice and terminates the TLS connection. The communication between the ingress gateway and microservices, and also between microservices is protected with mTLS. The Envoy proxy does JWT verification for Order Processing and Inventory microservices.

12.4.2 How to use JWT in service-to-service communication

In section 12.4.1 we protected both the Order Processing microservices and the Inventory microservice with JWT. For the Order Processing microservice, the client application has to send the JWT along with the HTTP request, then when the Order Processing microservice talks to the Inventory microservice, it passes the same JWT it got from the client application. (This is one of the use cases for securing microservices with JWT, which we discussed in detail in chapter 7 under section 7.1.1. You can also find the limitations of this approach in section 7.1.2.)

However, Istio does not support this use case at the time of this writing. Or in other words, Istio does not support propagating a JWT one microservice gets to another upstream microservice. So, while implementing this in our sample in section 12.4.1, we did that at the code level. We modified the Order Processing microservice to read the incoming JWT from the HTTP header (which Envoy passes through to the service behind, after verification) and attached it to the HTTP request, when the Order Processing microservice talks to the Inventory microservice. You can find the code with respect to this in

chapter12/services/order/src/main/java/com/manning/mss/ch12/order/client/InventoryClient.java class file.

12.4.3 A closer look at JSON Web Key (JWK)

In listing 12.10 we saw Envoy proxy uses a JWKS (JSON Web Key Set) endpoint to fetch a document, which carries a set of JSON Web Keys. A JWK is a JSON representation of a cryptographic key and a JWKS is a representation of multiple JWKs. The RFC 7517 (<https://tools.ietf.org/html/rfc7517>) defines the structure and the definition of a JWK. As per listing 12.10 the Envoy proxy gets the JWKS from <https://raw.githubusercontent.com/microservices-security-in-action/samples/master/chapter12/sample04/jwtkey.jwk>. Let's have a look at the content of this document (listing 12.11) and code annotations explain what each element means. If you'd like to delve more deeply into the details, please check RFC 7517. If you have a PEM encoded X509 certificate, you can use an online tool like <https://8gwifi.org/jwkconvertfunctions.jsp> to convert it to a JWK.

Listing 12.11. JSON Web Key Set (JWKS)

```
{
  "keys": [
    {
      "e": "AQAB",          #B
      "kid": "d7d87567-1840-4f45-9614-49071fc4d21",    #C
      "kty": "RSA",         #D
      "n": "-WcBjPsrFvG0wqVJd8vpV"      #E
    }
  ]
}
```

#A The parent element, which represents an array of JWKs.
#B A cryptographic parameter corresponding to the RSA algorithm.
#C The key identifier. This should match the kid value in the JWT header.
#D Defines the key type. The RFC 7518 defines the possible values.
#E A cryptographic parameter corresponding to the RSA algorithm.

12.5 Enforcing authorization

This section extends the use case we discussed in section 12.4 by enforcing authorization at the each microservice. We assume you have successfully completed all the samples we discussed in section 12.4. Here we use JWT to carry the end-user attributes, while using mTLS for service-to-service authentication – and enforce access control policies based on the attributes in the JWT.

12.5.1 A closer look at the JWT

Let's have closer look at the JWT you got from the Security Token Service (STS) while running the end-to-end sample in section 12.4. What you get from the STS is a base64-encoded string; you can use an online tool like <https://jwt.io> to decode it. The listing 12.12 shows the decoded JWT payload. In Istio we can define access control policies against any of these attributes.

Listing 12.12. The base64-decoded JWT payload

```
{
  "sub": "peter",      #A
  "aud": "*.ecomm.com",    #B
  "user_name": "peter",
  "scope": [      #C
    "foo"
  ],
  "iss": "sts.ecomm.com",    #D
  "exp": 1572480488,
  "iat": 1572474488,
  "authorities": [      #E
    "ROLE_USER"
  ],
  "jti": "88a5215f-7d7a-45f8-9632-ca78cbe05fde",
  "client_id": "applicationid"    #F
}
```

#A The user or the principal associated with this JWT – or the authenticated user.

#B Identifies the recipient(s) of the JWT.

#C Scopes associated with the JWT or what you can do with this JWT.

#D Identifies the issuer of the JWT.

#E The roles associated with the subject of the JWT.

#F Identifies the client application, which sends this JWT to the microservice.

12.5.2 Enforcing Role-base Access Control

In this section, we are going to define role-base access control (RBAC) policies for the Order Processing and Inventory microservices. In general terms, a *role* is a collection of permissions. A *permission* is a combination of a resource and an action. For example, the ability to do an HTTP GET on the Order Processing microservice is a permission. The HTTP GET is the action and the Order Processing microservice is the resource.

Similarly, the ability do a POST on the Order Processing microservice is another permission. Now we can combine these two permissions and can call it a role. Istio introduces a resource type called ServiceRole, which defines a set of rules. For example take a look at the ServiceRole definition in listing 12.13. There the services element represents a set of resources, and the methods element represents the allowed actions against those resources. In plain English, this says, if someone has the `order-viewer` role, he/she can do GET on the `order-service` Service running in the Kubernetes default namespace.

Listing 12.13. The order-viewer ServiceRole

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: order-viewer
spec:
  rules:
  - services: ["orders-service.default.svc.cluster.local"]    #A
    methods: ["GET"]    #B
```

#A An array of services, with the full qualified name, where this rule is applicable

#B An array of HTTP verbs, where this rule is applicable.

Let's take a look at another example in listing 12.14. It says, if someone has the `order-admin` role, he/she can do a POST on the `order-service` Service running in the Kubernetes default namespace.

Listing 12.14. The order-admin ServiceRole

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: order-admin
spec:
  rules:
    - services: ["orders-service.default.svc.cluster.local"]
      methods: ["POST"]
```

To attach or bind a ServiceRole to a user, Istio introduces a resource type called, `ServiceRoleBinding`. The listing 12.15 shows an example. There we bind (or map) a user having a role called `ROLE_USER` to the `order-viewer` ServiceRole. The way Istio finds the role of the user is by looking at the `authorities` attribute (which is a JSON array) from the JWT (listing 12.12). The issuer of the JWT or the STS, finds the roles of the user from an enterprise identity store (an LDAP, a database) connected to it and embeds those to the JWT under the attribute name `authorities`.

Listing 12.15. The order-viewer-binding ServiceRoleBinding

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: order-viewer-binding
spec:
  subjects:
    - properties:
        request.auth.claims[authorities]: "ROLE_USER"
  roleRef:
    kind: ServiceRole
    name: order-viewer
```

Once we have ServiceRoles and ServiceRoleBindings, to enable role-base access control (RBAC) in Istio, we need to define a `ClusterRbacConfig`. The `ClusterRbacConfig` in listing 12.16, enables role-base access control for all the Services under the Kubernetes default namespace.

Listing 12.16. The definition of the ClusterRbacConfig resource

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ClusterRbacConfig
metadata:
  name: default
spec:
  mode: ON_WITH_INCLUSION
  inclusion:
    namespaces: ["default"]      #A
```

```
#A Enables RBAC only for the Services in the Kubernetes default namespace
```

Istio defines four possible values for the mode element in listing 12.16.

- OFF: RBAC is completely disabled.
- ON: Enables RBAC for all the Services in the all the Kubernetes namespaces.
- ON_WITH_INCLUSION: Enables RBAC only for the Services in the Kubernetes namespaces mentioned under the inclusion element.
- ON_WITH_EXCLUSION: Enables RBAC for the Services in all the Kubernetes namespaces, except which are mentioned under the exclusion element.

Let's run the following command from chapter12/sample05 directory to create the ServiceRoles, ServiceRoleBinding and the ClusterRbacConfig as defined in listings 12.13, 12.14, 12.15, and 12.16 respectively.

```
\> kubectl apply -f .
clusterrbacconfig.rbac.istio.io/default created
servicerole.rbac.istio.io/order-viewer created
servicerole.rbac.istio.io/order-admin created
servicerolebinding.rbac.istio.io/order-viewer-binding created
```

To test the end-to-end flow after enforcing RBAC, please follow the steps defined in section 12.2.4. When you follow the steps in section 12.2.4, you should be able to do an HTTP GET to the Order Processing microservice, but will fail when doing an HTTP POST. The HTTP POST fails because the subject of JWT only carries the role ROLE_USER, and only a member of the order-viewer ServiceRole, not the order-admin ServiceRole.

12.6 Managing keys in Istio

In an Istio deployment, Citadel—the Istio control plane component—takes care of provisioning keys/certificates to each Envoy proxy. Also, it takes care of rotating keys/certificates. Citadel maintains an identity for each workload (or microservice) that runs under Istio and facilitates secure communication between workloads. When you enable mTLS in an Istio deployment, Envoy uses these provisioned keys to authenticate to each other. However, since Istio version 1.1, the way this works has changed significantly. In the following sections we discuss different approaches Istio uses to rotate keys/certificates.

12.6.1 Key rotation via volume mounts

Prior to Istio 1.1 or even with the latest Istio version with the default Istio profile (<https://istio.io/docs/setup/additional-setup/config-profiles>), provisioning of keys/certifies to Envoy proxies happen via Citadel, via volume mounts. The listing 12.17 shows the volume mounts associated with Envoy (or the Istio proxy), which runs in the Order Processing pod. If you run the command in listing 12.17, you need to replace the pod name (orders-deployment-f7bc58fbc-bbhwd) with the one that runs in your Kubernetes cluster. The Envoy proxy can access the certificates/keys provisioned to it by Citadel from the /etc/certs

location of its local file system. Citadel keeps track of the expiration time of each key/certificate it provisions and then rotates them before expiration.

Listing 12.17. The volume mounts of the istio ingress gateway pod

```
\> kubectl get pod orders-deployment-f7bc58fbc-bbhwd -o yaml
volumeMounts:
  - mountPath: /etc/istio/proxy
    name: istio-envoy
  - mountPath: /etc/certs/      #A
    name: istio-certs
    readOnly: true
  - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
    name: default-token-9h45q   #B
    readOnly: true
```

#A Mount path from the Envoy local file system for the keys used by Envoy proxy.

#B The default JWT, which is provisioned by Kubernetes – not specific to Istio.

Let's use the following command to print the PEM-encoded certificate provisioned to the Envoy proxy associated with the Order Processing microservice. Make sure to have the correct pod name corresponding to your deployment in the command.

```
\> kubectl exec -it orders-deployment-f7bc58fbc-bbhwd -c istio-proxy cat /etc/certs/cert-chain.pem
```

You can decode the output from the above command using an online tool like https://report-uri.com/home/pem_decoder – and the decoded output is shown in listing 12.18. There you can see the lifetime of the certificate is 90 days and it is issued for a service account.

Listing 12.18. The certificate provisioned to Envoy by Citadel

```
Issued By: cluster.local      #A
Serial Number: 3BF3584E3780C0C46B9731D561A42032      #B
Signature: sha256WithRSAEncryption      #C
Valid From: 12:37:06 27 Oct 2019
Valid To: 12:37:06 25 Jan 2020
Key Usage: Digital Signature, Key Encipherment      #D
Extended Key Usage: TLS Web Server Authentication, TLS Web Client Authentication      #E
Basic Constraints: CA:FALSE
Subject Alternative Names: URI:spiffe://cluster.local/ns/default/sa/default      #F
```

#A The issuer of the certificate

#B A unique number associated with this certificate

#C The CA signs the public key associated with this certificate following this signature algorithm

#D This certificate can be used to sign a message or encrypt a key. Cannot be used to encrypt data

#E This certificate can be used for mutual TLS authentication

#F The SPIFFE identifier corresponding to the service account associated with istio proxy.

When you create a pod in a Kubernetes environment you can associate a pod with a service account. There can be multiple pods associated with the same service account – and if you do not specify which service account you want to associate with a pod, then Kubernetes uses the default service account. In chapter 11, under the section 11.6, we discuss service accounts in

detail. Citadel adds the service account name to the certificate under the Subject Alternative Names field following the SPIFFE standard (for more information, see appendix I).

12.6.2 Limitations in key rotation via volume mounts

Since we use Istio add-on within GKE for all our samples in the book, the key rotation works in the way explained in section 12.6.1. But, this approach has some limitations or risks, including the following:

- Whenever Citadel rotates keys/certificates by updating the corresponding Kubernetes Secret, the corresponding Envoy proxies have to restart to load the new key/certificate.
- The private keys are generated by Citadel, and maintained outside the node, which runs the Envoy proxy, which uses those keys. There is a potential security issue that these keys can get compromised.

To overcome these limitations, since version 1.1, Istio introduced a Secret Discovery Service (SDS), which we discuss in section 12.6.3 in detail.

12.6.3 Key provisioning and rotation with SDS

In this section we discuss how key provisioning and rotation in Istio works with the secret discover service (SDS). Istio introduced the SDS feature with Istio 1.1 under the `sds` profile. So, if you are using Istio 1.1 or later, but still with the `default` profile, you will not see the SDS support. If you check the Istio feature status available on istio.io (<https://istio.io/about/feature-stages/#security-and-policy-enforcement>), you can find the status of the SDS feature. At the time of this writing, it is still in the alpha phase.

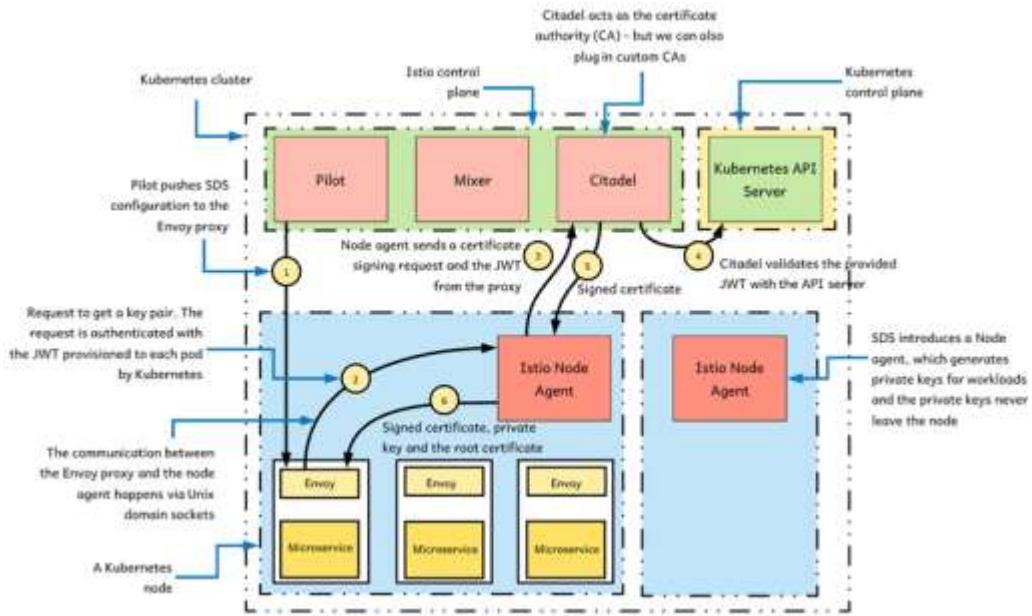


Figure 12.7 Secret Discovery Service (SDS) introduces a node agent, which runs on each Kubernetes node. The node agent generates a key pair for each workload (proxy) in the corresponding node and gets those signed by Citadel.

When you enable the SDS profile of Istio, you can find the SDS node agent running in every Kubernetes node under the `istio-system` namespace. The figure 12.7 explains how the Istio node agent (also known as the *SDS server*) facilitates key provisioning. Let's have a look at how each step works.

- Step-1: The Pilot, which runs on Istio control plane, keeps track of all the pods Kubernetes spins up. When it finds a new pod, Pilot talks to the corresponding Envoy proxy and passes the SDS configuration along with other configurations. For example, this includes how the Envoy proxy could talk to the node agent via Unix domain sockets (https://en.wikipedia.org/wiki/Unix_domain_socket).
- Step-2: The Envoy proxy talks to the node agent via Unix domain sockets to request a key pair. Along with this request, Envoy also passes the JWT provisioned to its pod by Kubernetes. This JWT is attached to the service account associated with the corresponding pod.
- Step-3: The node generates a key pair – and then a certificate signing request (CSR) – and sends the CSR along with the JWT it got from the proxy to Citadel. When generating the certificate, the node agent uses the service account name associated with the provided JWT to derive the Subject Alternative Name (listing 12.18) – and embeds that to the certificate. This generated certificate is compliant with SPIFFE standard, which we

discuss in appendix I.

- Step-4: Citadel talks to the API server to validate the provided JWT and confirms that this workload carries the correct JWT – and also running on the correct Kubernetes node.
- Step-5: Citadel returns back the signed certificate to the node agent.
- Step-6: The node agent returns back the signed certificate, the private key and the root certificate to the Envoy proxy (or the workload).

Once the keys are provisioned to workloads, the node agent keeps track of their expiration. It runs a timer job, which iterates over all the cached keys to find out the keys closer to expiry. If key is closer to expire, then the node agent sends a message to the corresponding Envoy proxy, and the proxy will start the same process as in figure 12.7 to get the new keys. One main advantage of using SDS to rotate keys is that we do not need to restart the Envoy proxy to reload new keys.

12.7 Summary

- A Service Mesh is an architectural pattern, which highlights the need of decoupling security, observability, routing control and resiliency from the microservices implementation to another level of abstraction.
- A service mesh operates in two planes in a microservices deployment: control plane and data place.
- The control plane manages the distributed network infrastructure used by the service mesh, while the data plane handles all runtime policies.
- Istio uses the Envoy service proxy, which is deployed alongside the microservice at the data plane to intercept ingress and egress traffic to and from a microservice and enforce security policies.
- Istio Citadel is control plane component, which provisions certificates to each service proxy in a microservices deployment and takes care of certificate rotation.
- Istio Mixer is control plane component, which defines and enforces authorization policies and throttling.
- The Istio ingress gateway runs at the edge of the Kubernetes cluster and terminates Transport Layer Security and can also establish an mTLS connection with upstream microservices.
- Istio introduces a new resource type called, DestinationRule. The DestinationRule is used to instruct Envoy proxies at the client side to use mTLS for all the communications with corresponding hosts.
- Istio introduces two custom resource types: ServiceRole and ServiceRoleBinding to perform authorization, and a ServiceRoleBinding is used to map a role/authority to a ServiceRole.
- Since Istio 1.1 onwards with the sds profile, Istio does key provisioning and rotation with the SDS.
- When you use SDS, the private keys associated with workloads never leave the

corresponding Kubernetes nodes – and also during key rotation we do not need to restart Envoy proxies to reload keys.

13

Secure coding practices and automation

This chapter covers

- OWASP top 10 API security vulnerabilities.
- Performing static analysis of code using SonarQube.
- Automating code analysis by integrating with Jenkins.
- Performing dynamic analysis of code using OWASP ZAP.

The complexity of the source code or the system design is a well-known source of security vulnerabilities. According to published research, after some point, the number of defects in an application increases as the square of the number of the lines of code. Unless you have good test coverage for both functionality and security tests, you won't be able to deploy changes into production frequently with confidence.

Two main kinds of security tests are integrated into the development lifecycle: static code analysis and dynamic testing. Both tests can be integrated into the build to run automatically after each daily build. One challenge with these tools is that they produce lots of false positives. In other words, most of the issues reported by these tools are invalid, but it takes many developer hours to go through the code to validate those issues. This problem is common in both open-source and commercial tools. One way to minimize the effect is to build a vulnerability management system that knows intelligently to reject any reported issues based on earlier experiences. If one issue is marked as invalid by one of your developers, the vulnerability management system learns from it, and when the same issue or a similar issue is reported later, the system automatically closes the issue as being invalid. In this chapter we

discuss how to engage security best practices in the microservices development lifecycle. If you are keen on reading more about application security, in general,

FURTHER READING If you'd like to learn more about security best practices, we recommend *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline* (O'Reilly Media, 2017) by Laura Bell, Michael Brunton-Spall, Rich Smith, and Jim Bird.

13.1 OWASP API security top 10

OWASP API security is an open source project (<https://github.com/OWASP/API-Security>) that is aimed at preventing organizations from deploying potentially vulnerable APIs. As we have discussed throughout this book, APIs expose microservices to consumers. It is therefore important to focus on how to make these APIs safer and avoid known security pitfalls. Let's take a look at the top 10 list of API security vulnerabilities.

13.1.1 Broken Object Level Authorization

Broken Object Level Authorization is a vulnerability that is present when using IDs to retrieve information from APIs. Users authenticate to APIs using protocols such as OAuth2.0. When retrieving data from APIs users may use IDs of objects to fetch data from. Let's take a look at an example API from Facebook where we get user details using an ID.

```
\> curl -i -X GET "https://graph.facebook.com/{user-id}
?fields=id,name&access_token={your-user-access-token}"
```

This is an example of an API that is used to retrieve details of a user identified by an id. We pass the user-id in the request as a path parameter to get details of the respective user. We also pass in the access token of the user who is authenticated to the API in a query parameter. Unless Facebook performs authorizations to check if the consumer of the API (the owner of the access token) has permissions to access details of the user to whom the ID belongs to, an attacker can gain access to details of any user they prefer—such as getting details of a user who is not in your friend list. This authorization check needs to happen for every API request.

To reduce this type of attack you should either avoid passing the user-id in the request or use a random (non-guessable) ID for your objects. In case your intention is to expose only the details of the user who is authenticating to the API through the access token you can totally remove the user identifier from the API and use an alternative identifier such as /me. For example,

```
\> curl -i -X GET "https://graph.facebook.com/me
?fields=id,name&access_token={your-user-access-token}"
```

In case you cannot omit passing in the user-id and need to allow getting details of different users, use a random non-guessable identifier for your users. Assume that your user identifiers were an auto-incrementing integer in your database. In a certain case you will be passing in something like the value 5 as the user and in another case something like 976. This hints the consumers of your API that you have user ids ranging from 5 to maybe something like a 1000

in your system and can therefore randomly request details of them. It is therefore best to use a non guessable ID in your system. In case your system is already built, and you cannot change its IDs, use a random identifier in your API layer and use an internal mapping system to map externally exposed random strings to the internal IDs. This way the actual ID of the object (user) remains hidden from the consumers of the API.

13.1.2 Broken Authentication

Broken authentication is a vulnerability that occurs when the authentication scheme of your APIs is not strong enough or is not implemented properly. Throughout this book we have learnt that OAuth2.0 is the de facto standard for securing APIs. OAuth2.0 combined with OIDC provides the required level of authentication and authorization for your APIs. We have however seen situations where API Keys (fixed keys) are used by applications to authenticate and authorize to APIs on behalf of users. This is mainly due to choosing convenience over security and therefore not a good practice at all.

OAuth2.0 works on opaque (random) access tokens or self-contained JWT-formatted tokens. As we discussed in chapter 3, when an opaque access token is used to access an API deployed on an API gateway the gateway will validate the token against the token issuer (STS). In chapter 7 we talked about JWTs and its attributes. If JWTs are used as access tokens the gateway can validate the token by itself. In either case the gateways need to make sure the authentication of the tokens is done properly. For example, in the case of JWTs, the gateways need to validate the tokens and check if,

- The issuer of the token is trusted (signature verified).
- The audience of the token is correct.
- The token is not expired.
- The scopes bound to the token permits it to access the resource being requested for.

Failure to implement the security scheme properly could lead to APIs being left vulnerable to attacks that could exploit them.

13.1.3 Excessive data exposure

APIs should return only the data that is relevant and required for its consumers. For example, if an application (consumer) requires whether a particular user is older than the age of 18, instead of exposing the user's date of birth or age the API should only return a Boolean which indicates the user is older than 18. This is also true for other software systems and websites. Software systems or websites should not expose the technology or versions of the technologies they run on. It's quite common to find technologies used in websites by viewing the HTML page source. If the website runs on a particular platform there are many cases where the JavaScript libraries or CSS that appear in the HTML source contain the names and versions of the technology platform. This is not a good practice since it allows attackers to mine for vulnerabilities of the mentioned technologies and attack the system using that information. It was not so long ago when an excessive data exposure vulnerability was uncovered in a mobile application called 3fun, which is a location-based online dating application. Using location data

that its API exposed unnecessarily, attackers could find dating preferences of the app's users in the White House, Supreme Court, and major cities in the world. By using the birthdate it exposed, attackers could pinpoint the exact users and hack into their personal photos.

13.1.4 Lack of resources and rate-limiting

APIs often do not impose limits on the number of requests they serve within a given time nor limit the amount of data returned. This can lead to attackers performing DDOS attacks that make the system unavailable to legitimate users. Imagine an API like the following that allows retrieving details of users:

```
https://findusers.com/api/v2?limit=10000000
```

If the API does not impose a limit on the maximum number of users that could be queried in a single API request, consumers could set a very large value on the limit. This would make the system fetch details of so many users that it would run the risk of consuming all resources it has and become unable to serve requests from legitimate users. A setting for the maximum number of records to be retrieved could be implemented at the application layer itself or by using an API gateway.

Similarly, attackers could perform DDOS attacks by sending a large number of requests within a very short time. For example, sending a million requests per second using distributed attack clients. This too would make the system unavailable for legitimate users. Preventing such attacks is typically done at the network perimeter using Web Application Firewall (WAF) solutions.

13.1.5 Broken Function Level Authorization

This vulnerability is about the lack of authorization at a fine-grained level of an API. An API can consist of multiple operations (resources)—for example, a `GET /users/{user-id}` for retrieving user information and `DELETE /users/{user-id}` for removing a particular user profile. Both operations are part of a single API called *users*. Authorization for this API should be done by operation, not only at an API level.

Performing authorization at an API level would result in anyone having permissions to retrieve user information (`GET`) and also being able to remove user information (`DELETE`), which is not correct. You could use OAuth2.0 scopes for this as we discussed in chapter 2. Different scopes could be bound to the different resources and only users with the relevant permission should be granted the scope when requesting OAuth2.0 access tokens.

There are cases where this authorization is delegated to the consuming application of this API (SPA). The SPA can use OIDC to obtain the roles of the user and hide the relevant action (`DELETE`) from the UI if the user doesn't have the proper role. This is not proper design since the functionality is still exposed at the API level and therefore remains vulnerable. There are also recommendations to make such operations that require fine-grained authorizations non-guessable, for example, using `GET /users/{user-id}?action=delete` instead of `DELETE /users/{user-id}`. This again is not good design. It not only messes up your API design, but it

doesn't solve the actual problem. It's not hard to find these links from various sources, and once they are discovered, your APIs would be vulnerable.

13.1.6 Mass Assignment

Mass assignment is a vulnerability that is exposed when APIs blindly bind to JSON objects received via clients without being selective about the attributes it binds to. Assume we have JSON that represents user attributes, including roles. A possible `GET /users/{user-id}` operation returns the following,

```
{"user": {
    "id": "18u-7uy-9j3",
    "username": "robert",
    "fullname": "Robert Smith",
    "roles": ["admin", "employee"]
}}
```

As you can observe this operation returns the details of the user including his/her roles. Now imagine using the same JSON to create or modify a user in the system. This is typically done by a `POST /users` or `PUT /users/{user-id}` to the users API and by passing in the JSON message. If the API assigns user roles by reading them from the JSON message that is passed in, anyone having permissions to add or modify users can assign roles to users, or even themselves. Imagine a Sign-Up form to a particular system being powered by such an API. This would enable anyone to sign themselves up as admin users in the system. To avoid such errors the API should be selective about what fields it picks from the input message to assign to its entities.

13.1.7 Security misconfigurations

Security misconfigurations on APIs can occur due to various reasons. The mainly occur due to insecure default configuration. For example, not disabling http (allowing only https) on your APIs, allowing unnecessary HTTP methods on API resources (allowing `POST` on a resource when only a `GET` is required), including stack traces on error messages that reveal internals of a system, permissive cross-origin-resource-sharing (CORS) that allows access to APIs from unnecessary domains and so on.

Preventing these types of error requires attention to both the design time of APIs and the runtime. We need to use tools which check our API specifications to make sure it adheres to API design best practices. This will prevent design time errors such as allowing unnecessary HTTP methods on APIs. Tools like the API auditor (<https://apisecurity.io/tools/audit/>) provided by APISecurity.io allow you to check your API definitions (Open API files) for vulnerabilities and malpractices in API design.

Runtime strengthening of the system needs to happen by automated mechanisms as much as possible. For example, instead of expecting an administrator to disable HTTP, the deployer scripts themselves should be automated to disable HTTP. In addition, the software should always be run on servers that have been hardened for security, and with all security patches

have been applied. You should also build strong verification systems (tests) that verify all necessary runtime configurations are applied. Netflix's Security Monkey (though now in maintenance mode) is one such tool developed to make sure their AWS and GCP accounts always run on secure configurations.

13.1.8 Injection

Injection flaws such as SQL injections, command injections, and so on can occur when APIs accept data and pass it on to interpreters to execute as a part of a query or command. For example, imagine a search operation on the users API that accepts a name to search users and passes it to a SQL statement. The API would look like the following:

```
GET /search/users?name=robert
```

The name received from the query would then be passed on to a SQL query that looks like this:

```
SELECT * FROM USERS WHERE NAME = robert;
```

Now if the name passed in is changed as "robert; DELETE FROM USERS WHERE ID = 1;" the resulting SQL statement would be as follows and remove a user from the system:

```
SELECT * FROM USERS WHERE NAME = robert; DELETE FROM USERS WHERE ID = 1;
```

To mitigate these types of attacks user input should always be sanitized. Static code analysis tools are also capable of detecting whether input parameters have directly been used in SQL statements. WAF solutions are also capable of detecting and preventing such attacks at runtime. Programming languages too have their own mechanisms for preventing such attacks. For example, Java provides the `PreparedStatement` construct, which can be used to execute SQL statements. It takes care of such vulnerabilities and prevents injection attacks.

13.1.9 Improper asset management

Platforms such as Kubernetes and containers have made it very easy for developers to deploy APIs into various environments. But this has brought a new challenge where a lot of APIs tend to get deployed easily and forgotten over time. When APIs are forgotten and newer versions of APIs deployed, the older versions get less attention. Organizations may miss applying the security patches and other fixes on old APIs, but they may still be in operation under the radar. Unless properly documented and maintained, people forget the details of these APIs and therefore unwilling to make changes to them, which could also lead to situations where they remain unpatched and vulnerable. It is therefore very important to document and maintain these APIs using a proper API Management system. These systems enforce best practices on APIs when being deployed and also give an indication of which APIs are being used and which are old enough to retire. These systems will also maintain the test scripts of the API and help you with testing the APIs when necessary.

13.1.10 Insufficient logging and monitoring

All actions performed in systems need to be logged, monitored, and analyzed for abnormalities. The lack of logs and monitoring results in not knowing what's going on in a system. Assume a user is accessing an API by using a token from an IP address from the United Kingdom. Now, if the same token is being used by a user from the United States a few minutes later, there should be something wrong going on. Our authentication and authorization layers won't detect anything wrong with these requests since they contain valid credentials to access APIs. We therefore need other mechanisms to detect such abnormalities. This is another instance where a proper API Management system can help. A system that analyses user behavior and processes them to find abnormalities and suspicious patterns are the only way of detecting and preventing such vulnerabilities.

13.2 Running static code analysis

In this section, we look at a practical sample of running a static code analysis using SonarQube (<https://www.sonarqube.org/>). SonarQube is an open source tool that helps you scan your code to check for security vulnerabilities, code smells, and bugs. It can be integrated with your build process so that it scans your code continuously (on each build). It can also be integrated with build automated tools such as Jenkins, which is something we will be looking at later in the chapter as well.

Static code analysis is a method of code debugging without executing the code itself (without running the program). Static analysis can help to check if the code adheres to industry best practices and prevent malpractices. Static analysis of code is important since it can reveal errors in code before an incident occurs by running the code after some time. Automated tools such as SonarQube can assist developers in performing static analysis on their code. Static analysis is however only the first step in software quality analysis. Dynamic analysis is also typically performed to ensure comprehensive coverage of code. This section of the chapter focuses on static code analysis; in section 13.4 we discuss dynamic code analysis as well.

The samples for this chapter can be found in <https://github.com/microservices-security-in-action/samples/tree/master/chapter13>. Download the code from this location to a directory of your choice. You will need Docker (<https://www.docker.com/>) installed on your machine to run these samples as well. We will be running SonarQube on a Docker container locally.

Run the following steps below to run the first sample of this section. Make sure you have your Docker process up and running. Running the following command from the command line in a terminal window prints the version of Docker that is running.

```
\> docker --version
```

Then open a terminal process and run the following command to download (pull) the docker image of SonarQube to your local working station:

```
\> docker pull owasp/sonarqube
```

Once this command is completed you have the SonarQube docker image on your local docker registry. The next step is to run the docker container by running the following command from your command line terminal.

```
\> docker run -d -p 9000:9000 -p 9092:9092 owasp/sonarqube
```

When this command completes you should see a random id printed to your console that indicates that SonarQube is now running on your machine. Running the following command gives you a list of running docker processes on your machine.

```
\> docker ps
```

You should see an image with the name 'owasp/sonarqube' running. You can now open a web browser and navigate to <http://localhost:9000> to open the SonarQube dashboard. If you haven't run any code scans yet you should see a message on the top right saying '0 projects scanned'.

The next step is to enable sonar scanning on our maven installation and run a scan of our code. To enable sonar scanning on your maven projects you need to add the following section to the `settings.xml` file located in your `$MAVEN_HOME/conf` directory (ex: `/usr/local/apache-maven-3.5.2/conf`). A sample `settings.xml` file can be found in the Github location of the samples of this chapter as well.

```
<settings>
  <pluginGroups>
    <pluginGroup>org.sonarsource.scanner.maven</pluginGroup>
  </pluginGroups>
  <profiles>
    <profile>
      <id>sonar</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <sonar.host.url>
          http://localhost:9000
        </sonar.host.url>
      </properties>
    </profile>
  </profiles>
</settings>
```

Once this is added to the `settings.xml` save and close the file and use your command line terminal to navigate to the `chapter13/sample01` directory. This directory contains source code from our Orders microservice sample from chapter 3. We are now about to scan this code to find potential security vulnerabilities in it. To do that run the following command at the prompt:

```
\> mvn clean verify sonar:sonar
```

Once the build succeeds use your web browser to navigate to <http://localhost:9000> (or refresh the page you visited previously). You should now see on top right of the page you have 1 project scanned. Click the number 1 to view details of the scanned project. A page appears that

says a project with name 'com.manning.mss.ch13.sample01' has been scanned. This project does not have any security vulnerabilities as of this writing. It reports 6 code smells. With newer versions of sonar, as newer vulnerabilities and malpractices are uncovered, this situation can change in the future. It is therefore possible for SonarQube to report vulnerabilities in this code as well at some point in time. Click the name of the project to view the details of the code smells. By navigating through these links, you can view the details of each code smell and figure out the causes of each. SonarQube explains the details of these nicely and provides recommendations of fixes as well.

Since the code did not have any interesting security vulnerabilities, let us now try out a sample that actually does. In sample 2 of this chapter we add a new microservice to our project which accepts credit card payments from users. This microservice has 1 method named 'pay'. The code of this service can be found in the sample2/src/main/java/com/manning/mss/ch13/sample02/service/PaymentService.java class. You will notice that this method has been annotated with the @RequestMapping("/payment") annotation. This annotation indicates that this operation is exposed over a path named /payment. Use your command line terminal tool and navigate to the chapter13/sample02 directory and execute the same command as before.

```
\> mvn clean verify sonar:sonar
```

Once the build completes refresh the SonarQube dashboard on your web browser. You would notice that the number of projects scanned has now increased to 2. Click it to view the details of the scanned projects. You should see the second project with name 'com.manning.mss.ch13.sample02' appearing on the project list. The project in sample 2 has 1 security vulnerability. Click the project name and then click the vulnerability to view details of it. You should see the following message:

```
Add a "method" parameter to this "@RequestMapping" annotation.
```

Clicking this message will take you to the exact point in code this vulnerability is detected. What SonarQube is reporting here is that our 'pay' method declares only the path of the operation (/payment) but it does not explicitly declare the http methods on which this operation is exposed.

Although our intention originally is to make the payments method available on http POST only, failure to declare this explicitly is now exposing this method over other unintended methods such as GET, DELETE and so on. This could potentially expose our system to users being able to perform things like, removing payment information from the system for example. This vulnerability falls under the API7 (security misconfiguration) category of the OWASP API Security top 10 vulnerabilities. Any API that exposes unnecessary http methods falls under this vulnerability category. It is therefore recommended to explicitly mention the method(s) on which this operation is being exposed. You can do that by changing the previous annotation as follows:

```
@RequestMapping(value = "/payment", method = RequestMethod.POST)
```

Note that you need to import the RequestMethod class by adding the following import statement to the top of the class, along with the other import statements.

```
import org.springframework.web.bind.annotation.RequestMethod;
```

Once the change is done, rerun the `mvn clean verify sonar:sonar` command from your command line prompt and observe the SonarQube dashboard. You will have to refresh the web page. You will now notice that the earlier reported vulnerability is no longer being reported.

13.3 Integrating security testing with Jenkins

At this point, you need to integrate security testing of our code with our build pipelines. You learned how to scan your code for vulnerabilities using Sonar and now you'll learn how to automate this process using Jenkins and why it's important to do so.

In most situations we work as teams and collaboratively work on projects where several people contribute code to a shared repository. In such situations it is not practical for us to depend on each individual to contribute bug and vulnerability free code. We need mechanisms that verify the code being contributed to alert when/if someone checks-in code with vulnerabilities. Identifying and preventing vulnerabilities as early as possible in our development lifecycle is when its easiest to fix them. Let us take a look at how we can setup Jenkins and configure a build pipeline.

13.3.1 Setting up and running Jenkins with a build pipeline

Jenkins is an open source automation server. It provides plugins to support many build, deployment and automation processes. In this section we will discuss how we can setup Jenkins on a Docker container and configure a simple build pipeline which compiles the source code of a microservice. You will need Docker to run the examples in this section. The code for the samples in this section are in <https://github.com/microservices-security-in-action/chapter13>. Note that this repository is not the same repository where we had code of other samples. The reason we need a separate repository for this is because Jenkins requires a configuration file (`Jenkinsfile`) to be present in the root of the repository. Having this code in a separate repository than the rest of the samples in this book makes it easier to teach and try out these samples. Check out the code from the repository and use the following instructions to set up Jenkins.

Make sure to have Docker running on your machine. Open up your command line tool and execute the command below to run the Docker image of Jenkins in a container.

```
\> docker run --rm -u root -p 7070:8080 -v jenkins-data:/var/jenkins_home -v /var/run/docker.sock:/var/run/docker.sock -v "$HOME":/home jenkinsci/blueocean
```

By `-p 7070:8080` we map port 7070 on our host (our machine) to the port of Jenkins (8080), which is running inside the container. The `-v "$HOME":/home` mounts the home directory of our host machine to path `/home` of the image within the container. If you are on Windows, the command should be as below. Note how the home path mount has changed.

```
\> docker run --rm -u root -p 7070:8080 -v jenkins-data:/var/jenkins_home -v
  /var/run/docker.sock:/var/run/docker.sock -v "%HOMEDRIVE%%HOMEPATH%":/home
jenkinsci/blueocean
```

This will start the Jenkins process on the terminal session you ran this on. You will see an id printed in-between two sets of asterisks in your command line output. This id is required for getting started with Jenkins. Once the process is started open your browser and navigate to <http://localhost:7070>. This will display the ‘Unlock Jenkins’ page and will prompt to enter the administrator password. Copy the id printed in-between the two sets of asterisks on your command line and paste it in the field that prompts for the admin password and proceed by clicking Continue.’

Next you will see the ‘Customize Jenkins’ page. Click the option to ‘Install suggested plugins’. This will install the default set of plugins recommended by Jenkins and may take a few minutes. Once the process is completed you will be prompted to create a new admin user. Provider details of an account you would like to create and proceed. You will now be directed to the home page of Jenkins. To install the SonarQube plugin click on the ‘Manage Jenkins’ link from the left menu and then click ‘Manage Plugins’ as shown in figure 13.1.

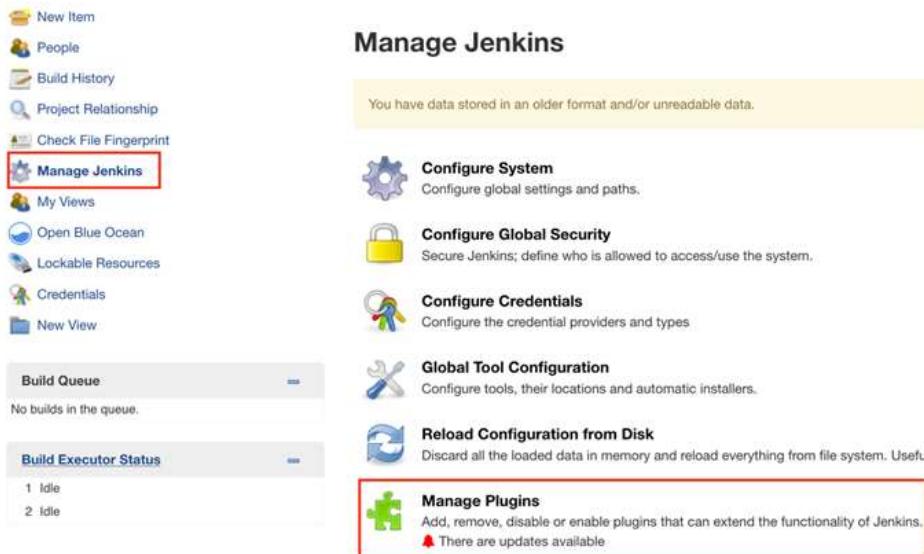


Figure 13.1 – The Manage Jenkins page to install plugins to Jenkins.

On the page that appears, click the ‘Available’ tab and search for ‘Sonar’ on the filter on the top right. Select the plugin named ‘SonarQube Scanner for Jenkins’ and then select ‘Install without restart’. Be sure to have your Sonar server running as instructed in section 13.2.

Once completed go back to the ‘Manage Jenkins’ page and click ‘Configure System’ to configure sonar for our build processes. You should now see a section named ‘SonarQube servers’. Click the ‘Add SonarQube’ button then fill in the details of the presented form as shown in figure 13.2. Provide ‘SonarScanner’ as the name and `http://host.docker.internal:9000` as the ‘Server URL’. Note that the reason we provide `host.docker.internal` as the host is because Jenkins is running within a Docker container and for it to connect to a port on the host machine `localhost` doesn’t work.

SonarQube servers

Environment variables

Enable injection of SonarQube server configuration as build environment variables
If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube installations

| | |
|-----------------------------|---|
| Name | SonarScanner |
| Server URL | <code>http://host.docker.internal:9000</code> |
| Server authentication token | - none - <input type="button" value="Add"/> |

Default is `http://localhost:9000`

SonarQube authentication token. Mandatory when anonymous access is disabled.

Figure 13.2 – Setting up the sonar configurations on Jenkins. Use `host.docker.internal` as the Sonar host URL.

Once completed press on the ‘Save’ button and proceed. The next step is to create our Jenkins build pipeline. The pipeline in Jenkins is the most important configuration which instructs Jenkins on the steps to perform when executing an automation process. To do that go to the Jenkins home page and click ‘New Item’ on the top left. Specify a name for your pipeline and select the ‘Pipeline’ option as shown in figure 13.3 and press on the ‘OK’ button on your bottom left.

Enter an item name

payments-service-pipe

Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Figure 13.3 – Creating a Jenkins pipeline for our project. Provide a name for your pipeline and press on the Pipeline option and proceed by clicking the Ok button on the bottom left.

Once this pipeline is created you will be navigated to the page where we can configure the newly created pipeline. Press on the 'Pipeline' tab as shown in figure 13.4. Specify 'Pipeline script from SCM' for the definition and select 'Git' as the SCM.

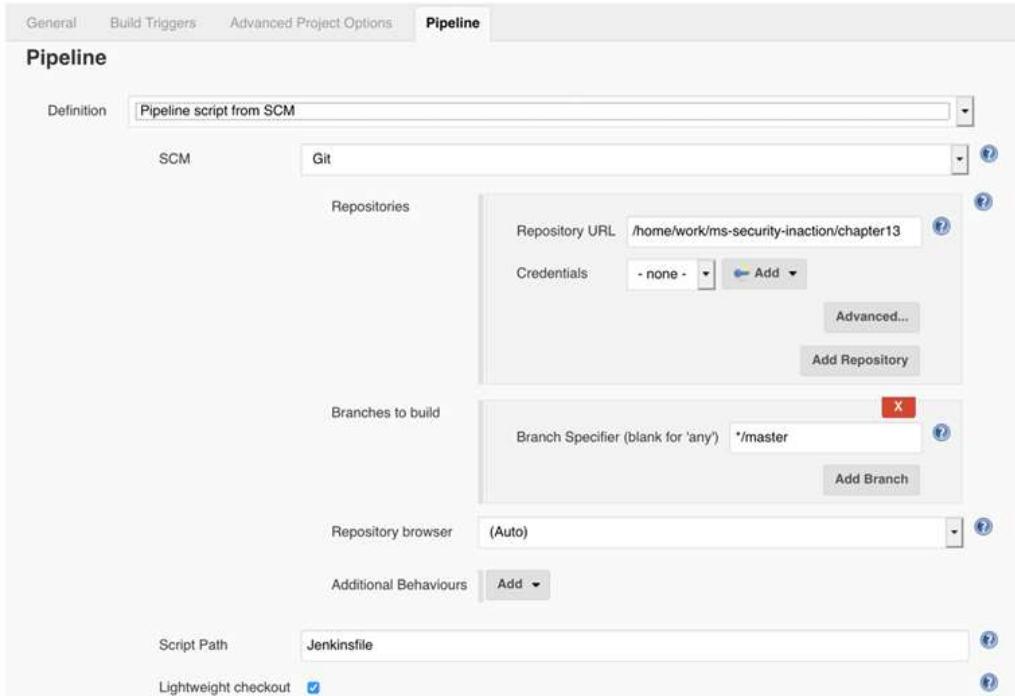


Figure 13.4 – The Jenkins pipeline configuration. Note how the path to the repository URL has been provided. Provided details as shown here and proceed by clicking the 'Save' button on the bottom left.

Note how the path to the 'Repository URL' has been provided. This should be the directory path where you cloned the repository containing samples of this section, where `/home` maps to `$HOME` path on your host machine. For example, if you cloned the repository to `/Users/roger/code/ms-security-inaction/chapter13` and `/Users/roger` is your `$HOME` path, the 'Repository URL' should be provided as `/home/code/ms-security-inaction/chapter13`.

Notice the 'Script Path' which specifies 'Jenkinsfile' as its value. This is the most important configuration file in this pipeline. This file should reside in the root of the mentioned 'Repository URL'. It contains the steps to execute in the pipeline. When you cloned the repository for this section you would have received a copy of this file as well. Let's take a look at its contents.

Listing 13.1 – Contents of the Jenkins file

```

pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('SonarQube analysis') {
            steps {
                withSonarQubeEnv(installationName: 'SonarScanner') {
                    sh 'mvn clean verify sonar:sonar'
                }
            }
        }
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
    }
}

```

The agent is typically a machine or container which executes tasks when instructed by Jenkins. In this case we use a docker image of maven to execute our build. This means that when you run your Jenkins pipeline, a docker image of maven will be executed.

The stages define the various phases in our Jenkins pipeline. In our case we have two stages, the first which runs the SonarQube analysis and the next stage which performs the build of our code. You can provide any name for the stage (`SonarQube analysis`). However, the `installationName` parameters should be the same as what you provided when configuring the SonarQube plugin for Jenkins (`SonarScanner`). The script provided within a step should be the same script we would execute when performing the relevant action manually. You can see that we execute the same instruction to scan our code for vulnerabilities from section 13.2 here as well. (`sh 'mvn clean verify sonar:sonar'`).

Once the mentioned details have been provided you can proceed to save the pipeline configuration. You should see your pipeline appearing on the home page of Jenkins. Click the pipeline and the click the 'Open Blue Ocean' link on the left. On the page that appears click on the 'Run' button to run your pipeline. Make sure to have SonarQube running before running the pipeline. You can click the relevant item to view the progress of your build. The very first execution of the pipeline can take a few minutes to complete, depending on the speed of your internet connection. This is because the maven container which runs to execute your build is brand new. It downloads all of the dependencies of your project to its local maven repository before it can execute the build. Builds after the first execution would be faster since by then the dependencies exist on the local maven repo of the container. However, since Jenkins too is running on a container, once the Jenkins container restarts the first build execute after that would be equal to our very first build as well.

You can view the progress of each stage. Clicking a particular stage will display the logs relevant to the action (script) being performed. Once the build is completed you should see the progress shown as in figure 13.5.



Figure 13.5 – The progress after the build is successful. You should see the result of your sonar scan by visiting the SonarQube application as mentioned in section 13.2.

You should now be able to visit the SonarQube application by pointing your browser to <http://localhost:9000>. The code we scanned is equal to that of sample02, but by fixing the vulnerabilities reported in it.

We just completed setting up our first Jenkins pipeline that scans our code using SonarQube and then builds it. We execute the pipeline manually by clicking the 'Run' button on the Jenkins pipeline. In ideal situations this pipeline will be run through automated processes. You can find the options you have to run this pipeline automatically by visiting the 'Build Trigger' section under the 'Configure' option of your pipeline. Jenkins allows us to run these pipelines using Github hooks¹, on periodic intervals and so on. We could also get our build job to fail if the relevant quality gates on SonarQube does not pass. This can be done by slightly advanced configurations on our Jenkinsfile as described in the SonarQube docs (<https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-jenkins/>). Jenkins could also be configured to trigger emails and other notifications when builds fail. This way we could configure fully automated build process for our projects that would also send out notifications upon failures.

13.4 Running dynamic analysis with OWASP ZAP

You can perform dynamic analysis on your applications using OWASP ZAP². ZAP, short for Zed Attack Proxy is an open source tool that helps to find security vulnerabilities in web applications.

Dynamic code analysis is a software testing mechanism that evaluates a program during real-time execution. Unlike static code analysis, the software needs to be run and used either

¹ <https://developer.github.com/webhooks/>

² https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

by a human or by automated scripts for tools to be able to perform dynamic code analysis. It's a process that tests various execution paths of a program by automatically generating various types of input parameters that would trigger different execution flows in it. Dynamic analysis is of great use to software developers and testers since it can increase the efficiency of the software testing process to a great extent.

ZAP is a tool that acts as a proxy between the client application (web browser) and server and analyses the requests and responses to identify potential vulnerabilities in the application. Figure 13.6 illustrates the said pattern.

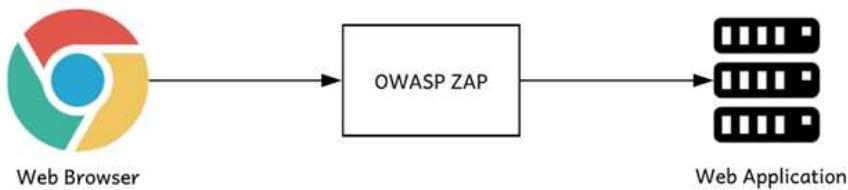


Figure 13.6 – OWASP ZAP acts as a proxy between the web browser and the web application. It intercepts all request and response exchanges between the two.

13.4.1 Passive Scanning versus Active Scanning

When it comes to penetration testing a passive scan is a harmless scan that looks only for the responses and check them against known vulnerabilities. It doesn't modify the website's data so it's safe to use against sites on which we don't have permission on. Passive scanning however is not every effective. Since it only looks at existing traffic and tries to identify threats and vulnerabilities by looking at passing data, the chances of detecting vulnerabilities are less. It can detect things like malpractices in data patterns, vulnerable libraries being used and so on.

Active scanning on the other hand is much more effective since it deliberately tries to penetrate the system using known techniques to find vulnerabilities. It does update the application's data. And it can insert malicious scripts into the system. It is therefore very important to perform active scanning tests only on systems you are permitted to. And they should be performed in dedicated environments so that they don't affect users of other environments (QA, Production).

13.4.2 Performing penetration tests with ZAP

We will use ZAP to perform a penetration test against an intentionally vulnerable website. You will need Java version 11+ to run the exercises in this section of the chapter. Download OWASP ZAP for your operating system from <https://github.com/zaproxy/zaproxy/wiki/Downloads> and install it. Run the software once installed. You will see a screen which asks to persist the ZAP session as in figure 13.7.

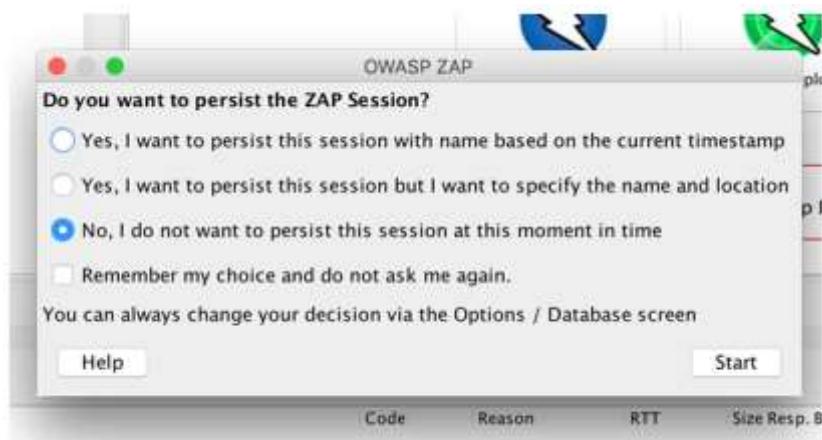


Figure 13.7 – ZAP start screen which asks to persist the ZAP session. We will not be requiring to persist the session for the exercises in this section.

Select the option to not persist the session and press the 'Start' button. Next, we need a web application to scan for. We will use the WebGoat application from OWASP for this. This is an open source web application that is purposely designed to contain vulnerabilities and very useful to learn about the various types of vulnerabilities in web applications. You can download the latest available version of the application from <https://github.com/WebGoat/WebGoat/releases>. At the point of this writing the latest version is v.8.0.0.M25 and that is what we will be running the exercises against. This application requires Java 11+. You can check the java version installed by running the command `java -version` in your command line too.

NOTE that this application contains many vulnerabilities and it is therefore recommended that you disconnect from the internet when running it.

Once you have downloaded the `webgoat-server-<version>.jar` file copy it to a preferred location and navigate to that location using your command line tool. Then execute the following command to run the WebGoat web application.

```
\> java -jar webgoat-server-8.0.0.M25.jar --server.port=9090 --server.address=localhost
```

This will start the WebGoat web application. You should see a message, which looks like below once the application has started successfully.

```
Started StartWebGoat in 14.386 seconds
```

Once started, open a web browser and navigate to `http://localhost:9090/WebGoat`. You should see a login screen and a link to register a new user. Click the link to create yourself an account

to use WebGoat and login. You could browse through the various links provided on the website. This is a website primarily designed by OWASP community for learning purposes and therefore it contains lots of valuable information and teaching material about the different types of attacks and remedies. Let us now look at how we can attack this site using ZAP.

Let's first perform an automated scan on the website to see if there are obvious misconfigurations or vulnerabilities. Go back to the ZAP UI and on the welcome page click the 'Automated Scan' option.



Figure 13.8 – The ZAP welcome page. Click the Automated Scan option to perform an automated scan on the WebGoat application.

On the next screen that appears, provide `http://localhost:9090/WebGoat` as the URL of the application to attack and select 'Use ajax spider' with 'jxBrowser' as shown in figure 13.9.

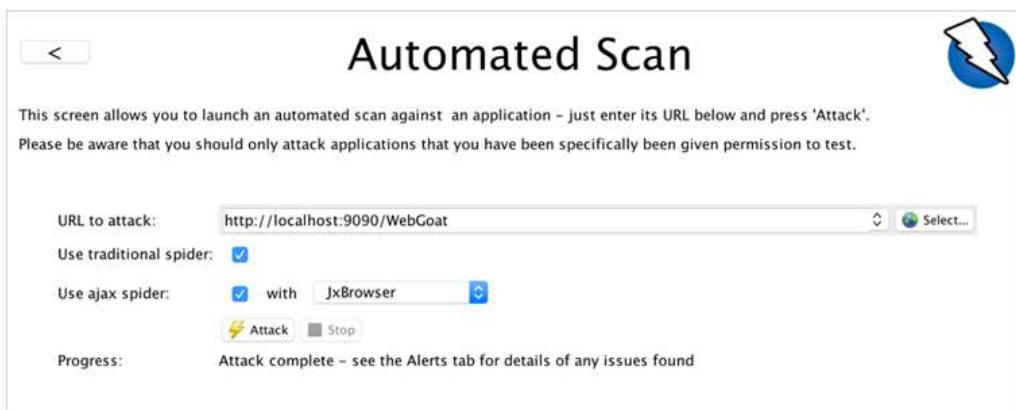


Figure 13.9 – Automated Scan screen in ZAP. Provide details of the WebGoat application and attack using the

jxBrowser.

Once the required information is provided press the 'Attack' button to start attacking the website. This will open up a web browser temporarily and run a series of attacks on the pages of the website. You can observe the progress of the attacks by clicking the 'Active Scan' tab. Once the attack is complete (progress is shown as 100%) check the 'Alerts' tab to view details of the discovered vulnerabilities and warnings. There should be a few yellow flags in the alerts section. You can click each of the alerts to discover it has reported. Since the WebGoat application requires a login to work, the automated scan cannot proceed beyond the login and user registration pages since those are the only pages accessible without a valid login session. This is true for many of the sensitive web applications in use. You can verify the details of the pages that were scanned by clicking the 'AJAX Spider' tab and observing the URLs of the scanned pages. You will notice that it mainly consists of links to CSS files, JavaScript files, images and paths of the login page and the user registration page. The automated scan therefore is not very effective when it comes to applications that require user logins to perform its actions.

For your penetration tests to be more effective you need to manually navigate the application in a way that covers as many features and combinations as possible. Let's navigate back the welcome page of ZAP by clicking the < button and then click the 'Manual Explore' option. Provide the URL of the WebGoat application to attack as <http://localhost:9090/WebGoat> and select Firefox as the browser to launch the application and click the 'Launch Browser' button as shown in figure 13.10.

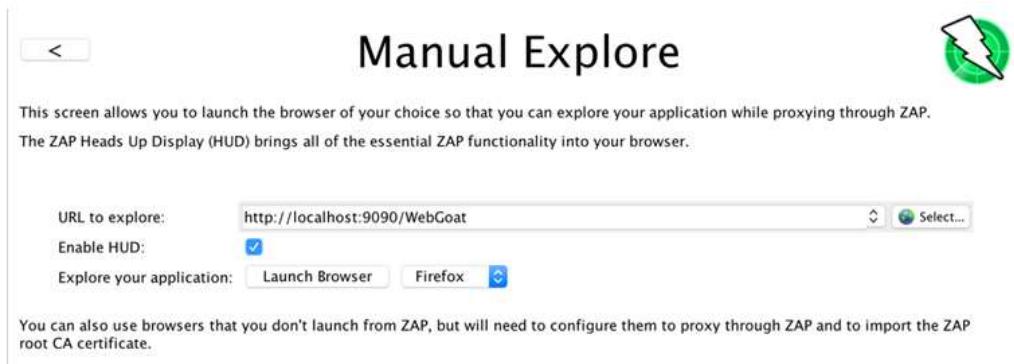


Figure 13.10 – The Manual Explore screen in ZAP. Provide details as shown in this figure and press on the Launch Browser button.

This will launch a Firefox browser and you will be presented with an option to either go through the HUD tutorials or continue to target. Select the option to continue to target. And once you do that will be presented with the screen to login to WebGoat. Login with the account you

created previously on WebGoat. Once logged in you will be presented with the home page of WebGoat as shown in figure 13.11.

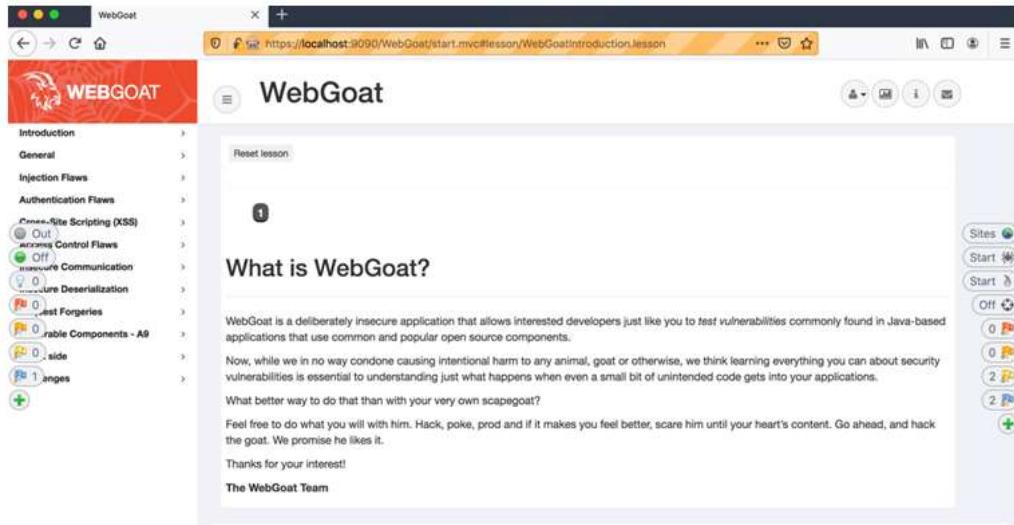


Figure 13.11 – Home page of WebGoat. The various tutorials of the website are navigable from the left menu.

We are now going to visit a vulnerable web page on this website and perform actions on it to see how ZAP identifies the vulnerabilities on it. Click the 'Cross Site Scripting (XSS)' link from the left menu and then click the first link that appears below that, 'Cross Site Scripting'. You will see a screen as shown in figure 13.12. Click the number '7' that appears on the menu on the top of the screen.

The screenshot shows a web browser window for the 'WebGoat' application. The URL is <https://localhost:9090/WebGoat/start.mvc?lesson=CrossSiteScripting.lesson>. The main content area is titled 'Cross Site Scripting'. It features a navigation bar with numbered steps (1 to 12), a 'Concept' section with a brief description, and a 'Goals' section listing learning objectives. On the left, there's a sidebar with a tree view of security flaws, including 'Cross-Site Scripting (XSS)' which is currently selected. On the right, there's a sidebar with various configuration buttons like 'Sites', 'Start', and 'Off'.

Figure 13.12 – Home page for Cross Site Scripting attacks on WebGoat. Navigating through these pages will teach you all about cross site scripting attacks and how to prevent them.

The 7th step in this tutorial is a page where you can try out a reflected XSS attack on a form. This page will have a form which allows you to checkout a few items from a shopping cart. Figure 13.13 shows what this form looks like.

The screenshot shows a 'Shopping Cart' page. At the top, it says 'Shopping Cart Items -- To Buy Now'. Below is a table showing four items in the cart:

| Item Description | Price | Quantity | Total |
|--|---------|----------|--------|
| Studio RTA - Laptop/Reading Cart with Tilting Surface - Cherry | 69.99 | 1 | \$0.00 |
| Dynex - Traditional Notebook Case | 27.99 | 1 | \$0.00 |
| Hewlett-Packard - Pavilion Notebook with Intel Centrino | 1599.99 | 1 | \$0.00 |
| 3 - Year Performance Service Plan \$1000 and Over | 299.99 | 1 | \$0.00 |

Below the table, it says 'The total charged to your credit card: \$0.00' and has a 'UpdateCart' button. There are fields for 'Enter your credit card number:' containing '4128 3214 0002 1999' and 'Enter your three digit access code:' containing '111'. At the bottom is a 'Purchase' button.

Figure 13.13 – The shopping cart form that is vulnerable to reflected XSS attacks. Press on the purchase button

to make ZAP detect the vulnerability.

Press on the 'Purchase' button. You will see a warning appear on the page informing about a cross site scripting attack. This is because ZAP detected that this page is vulnerable to reflected XSS once you submitted the form. To see the details of the vulnerability and see which field in the form is vulnerable, let's go back to the ZAP UI and click the 'Alerts' section. You should see a red flag on the alerts section which mentions about a cross site scripting vulnerability. If you click the relevant vulnerability you will be able to view details of it and also figure out which field in the form is vulnerable. Figure 13.14 illustrates what the ZAP UI look should like now.

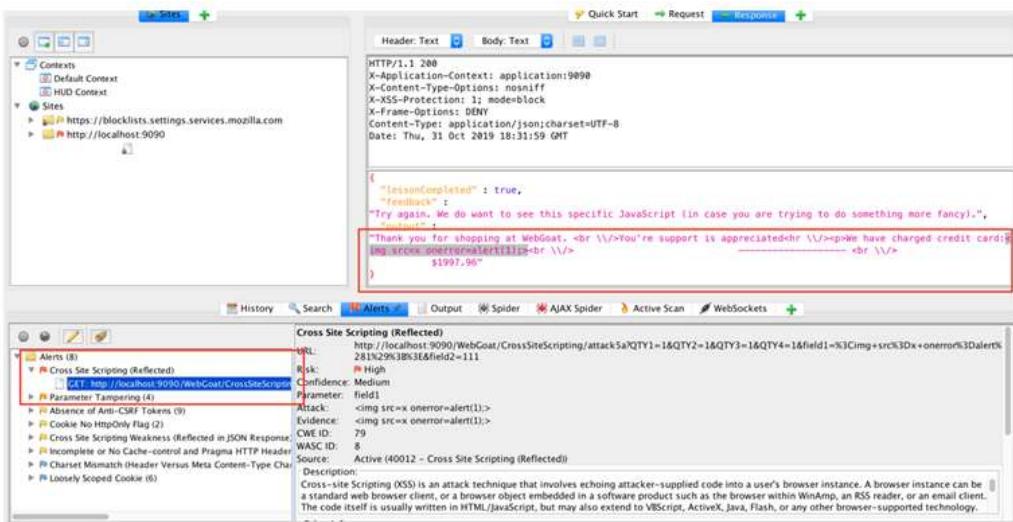


Figure 13.14 – ZAP reporting an XSS vulnerability on the page. It looks like the area on the page which prints out the credit card number is vulnerable.

If you take a look at the highlighted text box on top you will see that ZAP reports that an area in the page that seems to print out the credit card number when the 'Purchase' button was press is vulnerable to XSS. We can verify that by manually entering a JavaScript code into the field that accepts the credit card number. Go back to the Firefox browser from which you submitted the form and enter the following text in the 'credit card number' field and then press the 'Purchase' button.

```
<script>alert("Attacked");</script>
```

You should see that it pops up a JavaScript alert which says 'Attacked' as shown in figure 13.15.

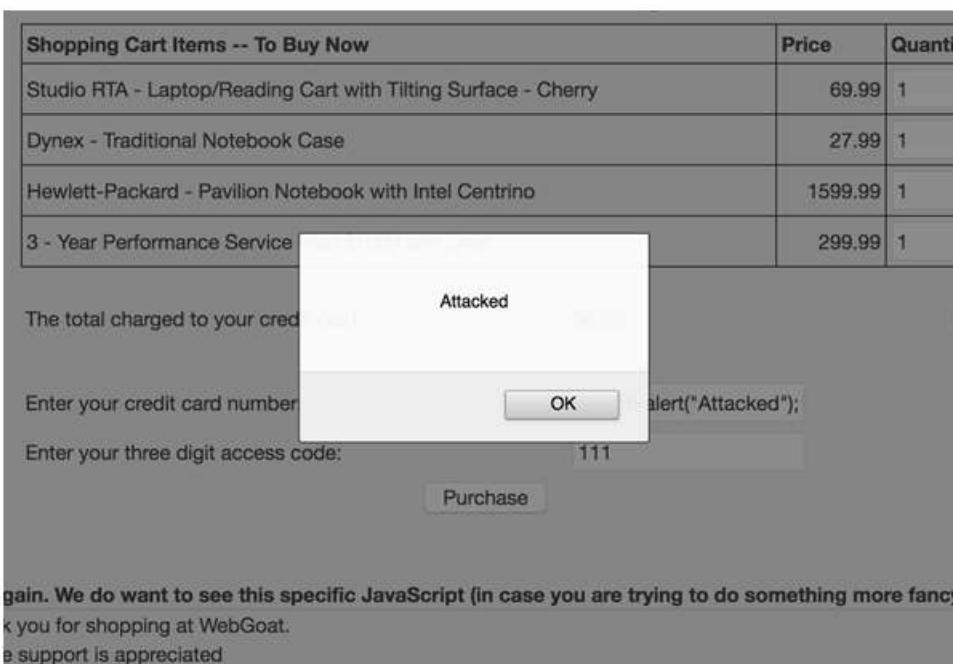


Figure 13.15 – The JavaScript popup that appeared when the purchase was attempted. It is clear that the page attempts to print whatever user input entered in the credit card number field.

ZAP can similarly be used to detect various kinds of dynamic attacks on web applications which cannot be identified by static code analysis tools. It also provides you recommendations for remedies in the tool itself.

13.5 Summary

- The OWASP top 10 API security vulnerabilities describe the most popular vulnerabilities discovered in APIs and recommended mitigation mechanisms for each of the identified vulnerability.
- You can use static analysis of code to debug code without execution to identify potential bugs and security vulnerabilities. SonarQube is an open source tool that can be used to scan code without execution to identify bugs and vulnerabilities in it.
- It is important to integrate our code scanning processes with automation tools such as Jenkins. It provides automated mechanisms of running code analysis through build pipelines and triggering notifications when failures occur, either due to code build failures or when the required quality gates of software do not pass.
- Dynamic analysis of code is a process that performs analysis of code while it is being executed through automated and manual processes. Dynamic analysis of code

generates different combinations of artificial inputs, which can test the different execution paths of code to identify bugs and vulnerabilities. OWASP ZAP is an open source tool that can be used to perform dynamic analysis of code.

A

Docker fundamentals

As a software developer, you probably have experienced the pain in distributing software you build and in some cases finding out that they don't work in certain environments. This is where the popular developer cry of ***it works on my machine*** is born! Docker helps you overcome this problem to some extent by packaging your software, along with all its dependencies, for distribution. In chapter 10 of the book, we discuss securing microservices deployed in a Docker environment. If you are new to Docker this appendix lays the right foundation you need in following chapter 10.

A.1 Docker overview

Docker is an open source project, which makes software packaging, distribution and running much simplified. It is also the name of a private company founded in 2010, which is behind the Docker open source project, and also maintains a commercial version of it. To avoid any confusion, whenever we talk about Docker - the company, instead of just calling it Docker we use Docker Inc. When we say just Docker, we mean the software produced by the Docker open source project.

Docker builds a layer of abstraction over the infrastructure (host machine). Any software that runs on Docker can be easily decoupled from the infrastructure and distributed. Docker's core capabilities are built on top of Linux kernel features. Linux kernel helps building an isolated environment for a running process. We call this isolated environment a container. A process running in a container has its own view of the file system, process identifiers, host name and domain name, network interface, and so on – and that does not conflict with the view of another process running in a container on the same host operating system. For example, two independent processes, each running in its own container, can listen on the same port, even though they run on the same host operating system. We learn more about the level of isolation containers bring in as we move forward in this appendix.

A.1.1 Containers prior to Docker

Docker brought containers into mainstream - but it is a concept few decades old (figure A.1). In 1979 we could change the root directory of a running process with the chroot system call introduced in Unix V7 and added in to Berkeley Software Distribution (BSD) in 1982. Chroot is still popular today to limit the visibility of the file system to a running process and considered as a best practice by system administrators. After almost two decades from chroot, FreeBSD added support for FreeBSD Jails in 2000. Jails built on top of chroot concept, allowed dividing a given host environment into multiple isolated partitions. Each of such partition is called a jail. A jail has its own set of users, and a process running in one jail cannot interact with another process running in another jail. Linux VServer followed a similar concept as in FreeBSD Jails and by 2001 it was possible to partition a Linux VServer, where each partition has its own, isolated file system, network and memory.

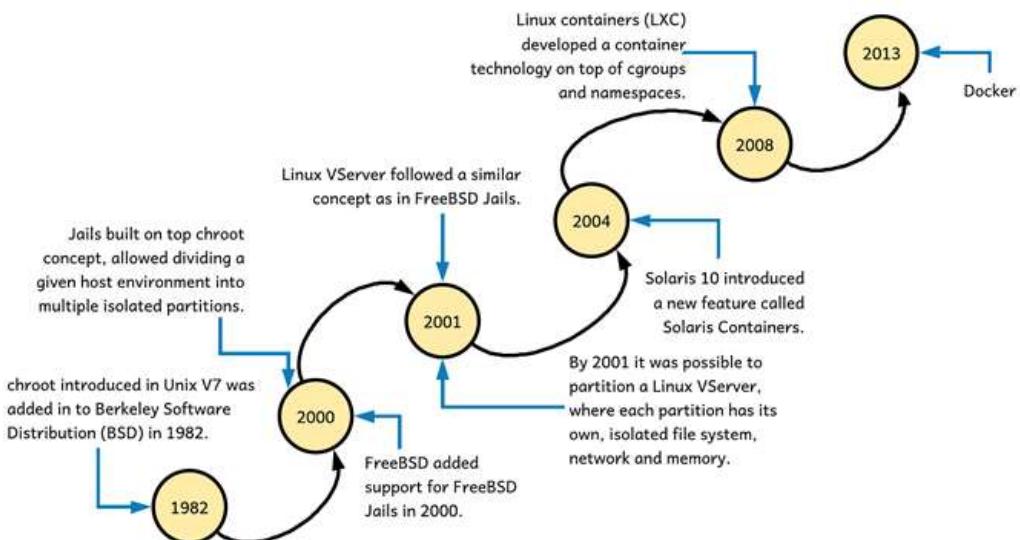


Figure A.1 The evolution of containers started in 1982 by introducing chroot to BSD.

In 2004, Solaris 10 introduced a new feature called Solaris Containers (also known as Solaris Zones). Like FreeBSD Jails, Solaris Containers too provided operating system-level virtualization. Google introduced Process Containers in 2006 as a way of building isolation over CPU, disk I/O, memory, and network; and a year later Process Containers was renamed and added to Linux Kernel 2.6.24 as control groups (cgroups). In 2008, Linux containers (LXC) developed a container technology on top of cgroups and namespaces. The cgroups and namespaces are fundamental to the containerization we see today. Docker up to version 1.10

was based on LXC (more details in section A.13). We further discuss cgroups and namespaces in detail and how they are related to Docker under sections A.13.4 and A.13.5.

A.1.2 Docker adding value to Linux containers (LXC)

The main value Docker provides over traditional Linux containers is the portability. It makes Linux containers portable between multiple platforms (not just Linux) and builds an ecosystem to share and distribute containers. Docker does this by defining a common format to package an application (or a microservice), with its all dependencies into a container. Developers can share this artifact on any platform, which runs Docker, and Docker makes sure it provides the same environment for it to run, irrespective of the underlying infrastructure.

A.1.3 Virtual machines vs. containers

A virtual machine provides a virtualized environment over the infrastructure. In general it operates on two modes: with a type-1 hypervisor and with a type-2 hypervisor. A hypervisor is software, which runs virtual machines – or in other words, manages the life cycle of a virtual machine. Virtual machines with type-2 hypervisor, is the most common model. When you run VirtualBox (<https://www.virtualbox.org/>) as a virtual machine (or even VMWare Player or Parallels Desktop for Mac), it operates with a type-2 hypervisor. As shown in figure A.2, the type-2 hypervisor runs on top of a host operating system – and all the virtual machines run on the hypervisor. Type-1 hypervisor does not require a host operating system – it directly runs on the physical machine. Any number of virtual machines can run on a hypervisor (subject to the resource availability of the host machine) and each virtual machine carries its own operating system. The applications running on different virtual machines (on the same host operating system) are isolated from each other, but the applications running on the same virtual machine are not.

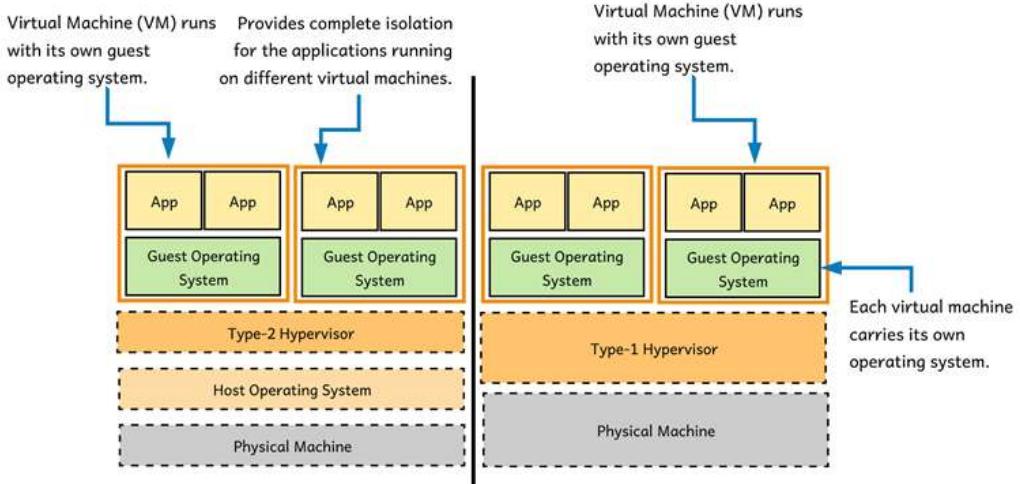


Figure A.2 A side by side comparison between type-2 hypervisor and type-1 hypervisor. Each virtual machine running on the host operating system carries its own guest operating system. The type-2 hypervisor provides an abstraction over the host operating system, while type-1 hypervisor provides an abstraction over the physical machine.

Unlike in a virtual machine, a container does not carry its own guest operating system; rather it shares the operating system kernel with the host. Figure A.3 shows that the containers with the applications, run on a Docker layer. But in reality, there is no Docker layer there – as the container itself is a construct of the Linux operating system. So the containers run on the kernel itself. Docker natively runs on Linux. To run Docker on other platforms we use a virtual machine with the Linux operating system and run Docker on top of that (we discuss this in detail under the section A.1.4). Since a container does not carry a guest operating system, but only the bare minimal software packages required to run the application, its far more lightweight than a virtual machine. That makes containers the much-preferred option to package and distribute microservices. Also, since a container has no guest operating system, the time it takes to boot up a container is far less than, that of a virtual machine.

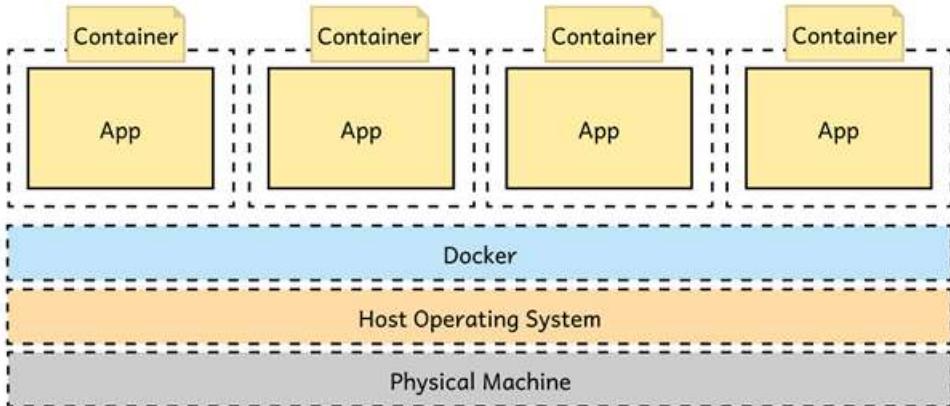


Figure A.3 Each container shares the operating system kernel with the host. Each application runs in its own container and the applications are isolated from each other.

In terms of packaging, we follow the pattern to have one application per container. In other words, a container represents a single process. Having one process per container helps us address scalability requirements much smoother. We cannot scale a process with the load, but the container. If we have multiple processes running in the same container, but with different scalability requirements, it would be hard to address that, just by scaling the container. Also, running multiple applications in a single container, defeats the purpose of a container, as there is no isolated environment for each application and further creates a management nightmare.

When we run multiple processes in a containerized environment, they cannot talk to each other directly as each process has its own container. To achieve the same level of isolation with a virtual machine, we need to package a virtual machine with each application, which creates a huge overhead on the host operating system.

A.1.4 Running Docker on non-Linux operating systems

When we run Docker on a non-Linux operating system such as Windows or Mac OS the layers of figure A.3 become a little different. Docker at its core uses basic Linux constructs to provide process isolation and it can run only on a Linux kernel. But, there is a work-around to make it work on non-Linux operating systems. There is some good history on how Docker support for Windows and Mac OS evolved over the past, but in this section we only focus on how it works at present. As shown in figure A.4, Docker for Mac OS uses xhyve hypervisor, based on Hyperkit (a tool kit for embedding hypervisor capabilities to an application on Mac OS) and Docker for Windows uses Hype-V hypervisor, which is built into Windows from Windows 10 onward. Both xhyve and Hype-V let you boot up Alpine Linux (a security oriented, lightweight Linux distribution) on Mac OS and Windows respectively and Docker runs on top of the Alpine Linux kernel.

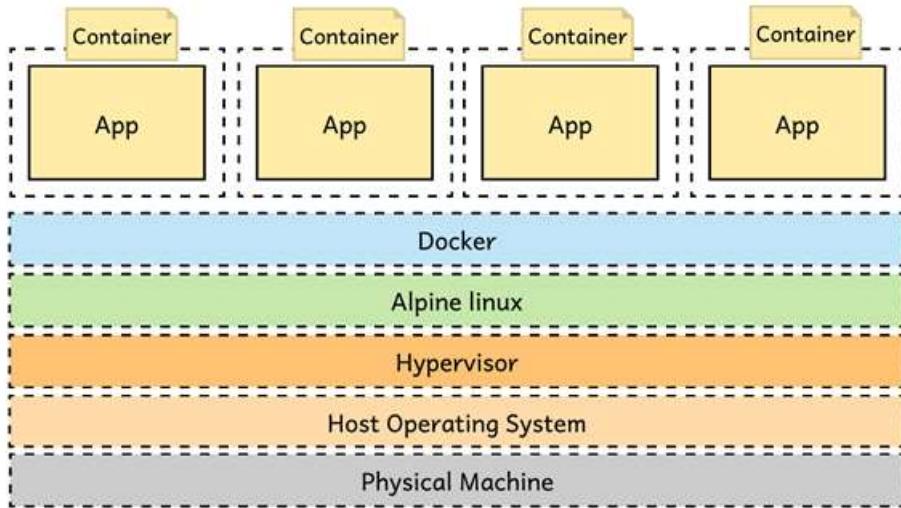


Figure A.4 Docker on non-Linux operating systems run on a hypervisor. Docker at its core uses basic Linux constructs to provide process isolation and it can run only on Linux kernel and in this case on Alpine Linux.

A.2 Installing Docker

Installing Docker on a local machine is not as hard as how it used to be in its early days. There are two versions of Docker: Community Edition (CE) and Enterprise Edition (EE). In this book we use Docker CE, which is free and open source. Docker EE and Docker CE both share the same core feature set, while Docker EE adds more on top of it at a cost, mostly targeting the large enterprises who run mission critical applications on Docker. We do not list Docker installation instructions in this section, mostly because those could change and there is a risk you end up with some stale instructions that do not work with the latest Docker version. Docker has clean documentation, explaining how to install Docker under different platforms. Please refer Docker documentation related to your platform from here (<https://docs.docker.com/install/#supported-platforms>) and follow the instructions to complete the installation. Once the installation is completed, run the following command to find out the details related to the Docker version you are running:

```
\> docker version
Client: Docker Engine - Community
Version:           18.09.1
API version:      1.39
Go version:       go1.10.6
Git commit:       4c52b90
Built:            Wed Jan  9 19:33:12 2019
OS/Arch:          darwin/amd64
Experimental:     false

Server: Docker Engine - Community
Engine:
```

```

Version:      18.09.1
API version: 1.39 (minimum version 1.12)
Go version:   go1.10.6
Git commit:   4c52b90
Built:        Wed Jan  9 19:41:49 2019
OS/Arch:      linux/amd64
Experimental: true

```

Docker Enterprise Edition (EE) vs. Docker Community Edition (CE)

Both the Docker EE and Docker CE share the same core, but Docker EE, which comes with a subscription fee, includes additional features such as private image management, container app management, cluster management support for Kubernetes and Swarm (which we talk about in chapter 11) and integrated image security scanning and signing. Explaining each of these features is out of the scope of this book. Anyway, if you are keen on learning Docker in detail, we recommend you having a look at the books *Docker in Action* (Manning Publications, 2019) by Jeff Nickoloff and Stephen Kuenzli, and *Docker in Practice* (Manning Publications, 2019) by Ian Miell and Aidan Hobson Sayers. Also, the book *Docker Deep Dive* (independently published, 2018) by Nigel Poulton gives a very good overview of Docker internals.

A.3 Docker high-level architecture

Before we delve deep into Docker internals and how it builds a containerized environment, lets first have a look at the high-level component architecture. Docker follows a client-server architecture model as shown in figure A.5. The Docker client talks to the Docker daemon running on the Docker host over a REST API to perform various operations on Docker images and containers (we discuss the difference between an image and a container later in this section – and for now think that a Docker image is the distribution unit of your application and the container is the running instance of it). Docker daemon supports listening on three types of sockets: Unix, TCP and FD (file descriptor). Only by enabling TCP socket will let your Docker client talks to the daemon remotely. All the examples in this appendix assume, the communication between the client and daemon happens over the Unix socket (the default behavior), so you run both the client and the daemon on the same machine. In chapter 10, under section 10.7, we discuss how to enable remote access to Docker daemon over TCP and secure Docker APIs.

To run an application as a Docker container, we execute the `docker run` command via the Docker client (see step-1 in figure A.5). Docker client creates an API request and talks to the Docker daemon running on Docker host (step-2). Docker daemon checks whether the Docker image requested by the client is present locally and if not, it talks to a Docker registry (a store of Docker images) and pulls (step-3 and step-4) the corresponding image (and all its dependencies) and starts running it as a container (step-5).

A Docker image is the packaging of your application. To share a Docker image with the developers or with the public, we can use a Docker registry. For example, Docker Hub (<https://hub.docker.com>) is a public Docker registry. A container is a running process. We start

a container using an image and the same image can be used to run multiple containers. In other words, a container is a running instance of an image.

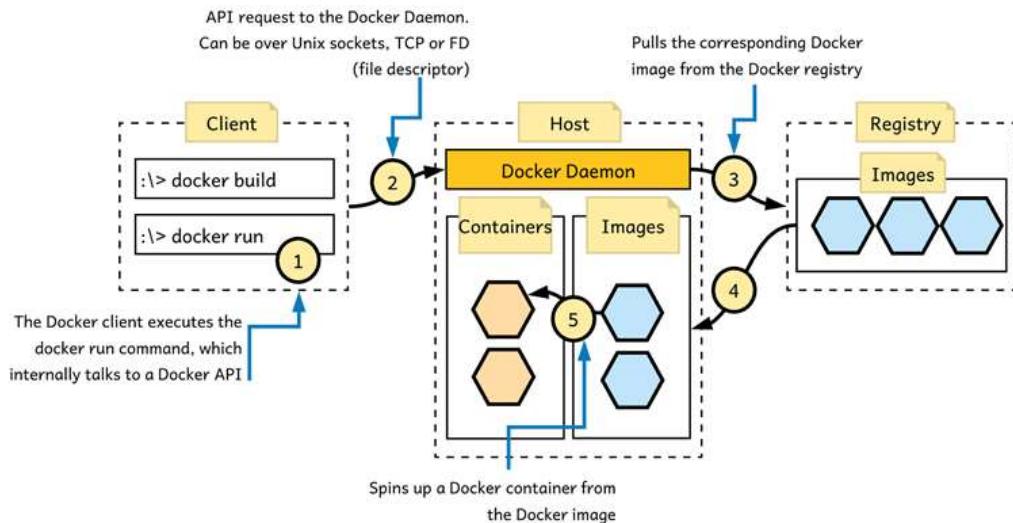


Figure A.5 High-level Docker component architecture. The Docker client talks to the Docker daemon running on the Docker host over a REST API to perform various operations on Docker images and containers.

The following command instructs Docker to pull the hello-world image from the Docker Hub (when you run it for the first time) and run it as a container. The process running inside the Docker container prints the message, Hello from Docker and the rest of the text after that:

```
\> docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:2557e3c07ed1e38f26e389462d03ed943586f744621577a99efb77324b0fe535
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit: <https://docs.docker.com/get-started/>

A.4 Containerizing an application

In chapter 7 we learn about JSON Web Tokens (JWT) and how to secure microservices with a JWT issued by a Security Token Service (STS). In chapter 10, we revisit the same use case, but in a containerized environment. Before running an application (in this case the STS, which is a Spring Boot application developed in Java), in containerized environment first we need to create a Docker image. In this section we are going to create a Docker image for the STS.

A.4.1 What is a Docker image?

A Docker image is a file, which packs your software for distribution, which is built with multiple layers (we discuss layers in section A.9). If you are familiar with object-oriented programming (OOP), you know about classes and objects. An image in Docker is analogous to a class in OOP. Like you can create multiple objects or instances from a given class, a running instance of a Docker image is called a container. We can create multiple Docker containers from a single image.

A.4.2 Building the application

The source code related to all the samples in this appendix is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, inside the appendix-a directory. The source code of the STS is available under appendix-a/sample01 directory. Run the following Maven command from the appendix-a/sample01 directory to build the STS. If everything goes well, you should see the BUILD SUCCESS message at the end:

```
\> mvn clean install  
[INFO] BUILD SUCCESS
```

Now if you look at the appendix-a/sample01/target directory, you will find a jar file, which is the security token service, which we just built.

A.4.3 Creating a Dockerfile

To run the security token service in a Docker container, we need to build a Docker image from the jar file we created in section A.4.2. First step in building a Docker image is to create a Dockerfile (see listing A.1). A Dockerfile includes step-by-step instructions for Docker, on how to create a Docker image. Let's have a look at the following Dockerfile (also available at appendix-a/sample01 directory), which instructs Docker to create an image for the security token service (which we built in section A.4.2) with the required dependencies:

Listing A.1. The content of the Dockerfile

```
FROM openjdk:8-jdk-alpine    #A
    ADD target/com.manning.mss.appendixa.sample01-1.0.0.jar /com.manning.mss.
        appendixa.sample01-1.0.0.jar    #B
    ADD keystores/keystore.jks /opt/keystore.jks    #C
ADD keystores/jwt.jks /opt/jwt.jks    #D
ENTRYPOINT ["java", "-jar", "com.manning.mss.appendixa.sample01-1.0.0.jar"]    #E

#A 1
#B 2
#C 3
#D 4
#E 5
```

The 1st line of the Dockerfile in listing A.1, instructs Docker to fetch the Docker image called, `openjdk:8-jdk-alpine` from Docker registry, and in this case from the public Docker Hub, which is the default option. This is the base image of the Docker image we are about to create. When we create a Docker image, we do not need to create it from scratch. If there are any other Docker images already available for the dependencies of our application, we can simply reuse them. For example, in this case to run our application, we need Java, so we start building our image from an existing OpenJDK Docker image, which is already available in Docker Hub.

The 2nd line instructs Docker to copy the file `com.manning.mss.appendixa.sample01-1.0.0.jar` from the `target` directory of the host file system to the root of the container file system. The 3rd line instructs Docker to copy `keystore.jks` file from the `keystores` directory of the host file system to the `/opt` directory of the container file system. This is the keystore security token service uses to enable Transport Layer Security (TLS). The 4th line instructs Docker to copy `jwt.jks` file from the `keystores` directory of the host file system to the `/opt` directory of the container file system. This keystore contains the private key, which STS uses to sign JWTs it issues. Finally the 5th line instructs Docker the entry point to the container – or which command to run when we start the container. For example in this case, Docker executes the `com.manning.mss.appendixa.sample01-1.0.0.jar` file.

A.4.4 Building a Docker image

The following command, run from `appendixa/sample01/` directory, instructs Docker to use the Dockerfile (see listing A.1) from the current path and build a Docker image out of it. Before executing the command, make sure that you have Docker up and running in your machine¹:

```
\> docker build -t com.manning.mss.appendixa.sample01 .
```

In the above command, we do not need to specifically mention the name of the Dockerfile. If we just leave it blank or if we do not specify a file name, Docker by default looks for a file with the name `Dockerfile` at the current location. The `-t` option in the above is used to specify the

¹ If you have Docker running in your local machine, `docker version` command should return a meaningful output with the proper versions of Docker Engine client and server.

name for the Docker image, in this case: `com.manning.mss.appendixa.sample01`. Following shows the output of the above command:

```
Sending build context to Docker daemon 22.32MB
Step 1/5 : FROM openjdk:8-jdk-alpine
8-jdk-alpine: Pulling from library/openjdk
e7c96db7181b: Pull complete
f910a506b6cb: Pull complete
c2274a1a0e27: Pull complete
Digest: sha256:94792824df2df33402f201713f932b58cb9de94a0cd524164a0f2283343547b3
Status: Downloaded newer image for openjdk:8-jdk-alpine
--> a3562aa0b991
Step 2/5 : ADD target/com.manning.mss.appendixa.sample01-1.0.0.jar
/com.manning.mss.appendixa.sample01-1.0.0.jar
--> 802dd9300b9f
Step 3/5 : ADD keystores/keystore.jks /opt/keystore.jks
--> 125e96cbc5a8
Step 4/5 : ADD keystores/jwt.jks /opt/jwt.jks
--> feeb468921c9
Step 5/5 : ENTRYPOINT ["java", "-jar", "com.manning.mss.appendixa.sample01-1.0.0.jar"]
--> Running in 5c1931cc19a2
Removing intermediate container 5c1931cc19a2
--> defa4cc5639e
Successfully built defa4cc5639e
Successfully tagged com.manning.mss.appendixa.sample01:latest
```

Here you can see, from the above output, Docker executes each line in Dockerfile in steps. Each instruction in the Dockerfile adds a read-only layer to the Docker image, except few specific instructions. We learn about image layers, later in this appendix under section A.9. The following Docker command lists out all the docker images in your machine (or the Docker host):

| \\> docker images | | | | |
|-----------------------------------|--------|--------------|-------------|-------|
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
| com.manning.ms.appendixa.sample01 | latest | defa4cc5639e | 11 mnts ago | 127MB |
| openjdk | | 792ff45a2a17 | 3 weeks ago | 105MB |

A.4.5 Running a container from a Docker image

Now we have a Docker image built for our security token service. If we would like, we can publish it to the public Docker registry or to a private registry, so that others can use it as well. Before we do that let's try to run it locally and see how we can get a token from the containerized security token service. Run the following command using the Docker client from anywhere in the machine you built the Docker image (to be precise you run the Docker client from a machine, which is connected to the Docker host, which has the image you built).

```
\\> docker run -p 8443:8443 com.manning.mss.appendixa.sample01
```

The above command will spin up a Docker container from the image (`com.manning.mss.appendixa.sample01`) we created in the section A.4.4 and start the security token service on the container port 8443 and map it to the host port 8443. This port mapping is done by the `-p` argument we pass to the `docker run` command, where the first

8443 in the command represents the container port and other the host port. Also make sure that there is no other process running in your host machine on port 8443. We will learn why we have to do the port mapping under the section A.18, when we talk about Docker networking. Once we start the container successfully, we see the following logs printed on the terminal:

```
INFO 30901 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
  8443 (https)
INFO 30901 --- [main] c.m.m.appendixa.sample01.TokenService : Started TokenService in
  4.729 seconds (JVM running for 7.082)
```

Now let's test the security token service, with the following cURL command. This is exactly the same cURL command we used in chapter 7, under the section 7.6.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type:
  application/x-www-form-urlencoded; charset=UTF-8" -k -d
  "grant_type=password&username=peter&password=peter123&scope=foo"
  https://localhost:8443/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works fine, the security token service returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIkpxVCJ9.eyJleHAiOiE1NTEzMTIzMzYsInVzZXJfbmFtZSI6InBldGVyIiwiYXV0aG9yaXrpZXMiOlsluk9MRV9VU0VSI1osImp0aS16jRkMmjNj04LTQ2MWQtNGV1Yy1hZT1jLTV1YWUxZjA4ZTJhMiIsImNsawVuDF9pZC16ImFwcGxpY2F0aW9uaWQiLCJzY29wZSI6WyJmb28iXX0.tr4yu mGLtsH7q9Ge2i7gxyTs0o0a0RS0Yoc2uBuAW50VKZcVsIITWV3bDN0FVHBzimpAPy33tvicFR0hBFoVThqKXzzG 00SkURN5bnQ4ufLAP0NpZ6BuDjvvVmwxNxrQp21VX141Q4eTvuyZozjUSCxzC1LNw5EFFi22J73g1_mRm2j- dEhbP1TvMaRKLBdk2hzIDVKzu5oj_gODBFm3a- IJjYoCimIm2igcesXkipRjtNcrJSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_ K97j1T6Rn1XZB1veyjfKs_XIXgu5qizRm9mt5xg", "token_type": "bearer", "refresh_token": "", "expires_in": 5999, "scope": "foo", "jti": "4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

A.5 Container name and container id

When we start a container from `docker run` command, we have the option to specify a name for the container by passing the `name` argument. In the following command, we start a Docker container from the `hello-world` image, with the container name, `my-hello-world`.

```
\> docker run --name my-hello-world hello-world
```

In case we skip the `name` argument, then Docker will assign a generated UUID as the name of the container. In addition to the name, a container also carries an id (container id) – and it is a randomly generated identifier, which cannot be changed. The main difference between the container id and the name is, we can change the name of a running container (using `docker rename`) – but the container ids are immutable. It is always better to give a container a name, which is human readable or easy to remember. That helps you performing certain operations on a running container. Otherwise, if you want to do something on a container, first you need to find out the corresponding container id or the system generated container name.

A.6 Docker registry

A Docker registry is the place where we store Docker images for distribution. It is a storage and content delivery system for Docker images. A registry has multiple repositories. A repository is a collection of different versions (or tags) of a given Docker image (figure A.6). To deploy Docker registry locally, in your environment, you can follow the steps defined in Docker online documentation: <https://docs.docker.com/registry/deploying/>.

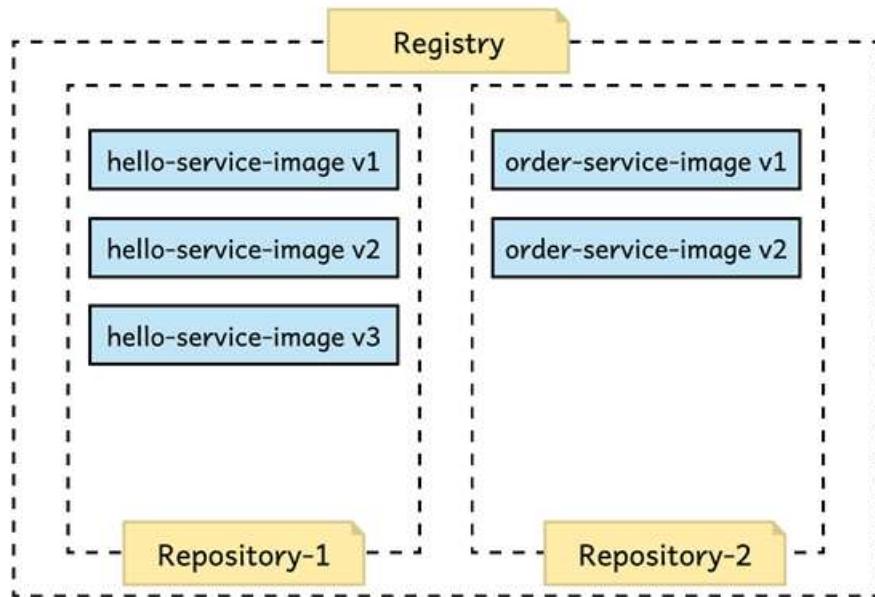


Figure A.6 A Docker registry has multiple repositories and each repository has multiple versions of a Docker image.

There are two versions of Docker registry, which you can deploy locally. One is the open source, community version, and the other is Docker Trusted Registry (DTR) from Docker Inc. DTR is not free and the cost comes with a set of new additional features, such as, built-in access control (along with LDAP/Active Directory integration), security scanning of images and image signing. You can refer Docker online documentation to find out more about DTR: <https://docs.docker.com/ee/dtr/>.

A.6.1 Docker Hub

Docker Hub (<https://hub.docker.com>) is the best-known Docker registry – and also the default registry in Docker. It's a hosted service provided by Docker Inc. Docker Hub offers both public and private repositories. Public repositories are free and an image published to a public

repository is accessible to anyone. At the time of writing, Docker Hub only offers one free private repository – and if you need more, you have to pay.

A.6.2 Harbor

Similar to Docker Trusted Registry (DTR), Harbor (<https://goharbor.io/>) is a registry, which is also built on top of open source Docker registry, with some additional features (mostly security and identity management), but unlike DTR, Harbor is open source.

A.6.3 Docker cloud platforms and registries

Instead of running Docker on your own servers and maintaining the hardware yourself, you can look for a cloud vendor who is providing a Docker platform as a service. There are multiple cloud vendors doing that. All these vendors provide their own Docker registries too – as a hosted service.

- Google Container Registry (GCR) integrates with Google Cloud Platform.
- Amazon Elastic Container Registry (ECR) integrates with Amazon Elastic Container Service (ECS).
- Azure Container Registry (ACR) by Microsoft integrates with Azure Container Service (ACS).
- OpenShift Container Registry (OCR) by Red Hat integrates with OpenShift Container Platform (OCP).
- Pivotal Container Service integrates Harbor as the Docker registry.
- IBM Cloud Container Registry integrates with IBM Cloud Kubernetes Service (we discuss Kubernetes in appendix B).
- Oracle Container Registry integrates with Oracle Container Engine for Kubernetes.
- VMware Harbor Registry integrates with VMware Cloud PKS.

A.7 Publishing to Docker Hub

In this section we'll see how to publish the Docker image we created in section A.4.4 to the public Docker registry, which is the Docker Hub. First we need to create a Docker ID from <https://hub.docker.com>. In the following section, we use our Docker ID, prabath; you need to replace it with your own Docker ID. Let's use the following command to create a valid login session with Docker Hub and then enter the corresponding password:

```
\> docker login --username=prabath
Password:
Login Succeeded
```

Next, we need to find the image ID of the Docker image we need to publish to Docker Hub. Following command lists out all the images in the Docker host machine and we can pick the image ID corresponding to our Docker image:

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------------------------------|--------|--------------|-------------|-------|
| com.manning.mss.appendixa.sample01 | latest | defa4cc5639e | An hour ago | 127MB |

Now we need to tag the image with our Docker ID as shown in the following command and you would need to use your own Docker ID there. We discuss tagging in section A.8. Here e7090e36543b is the image ID and prabath/manning-sts-appendix-a is the name of the Docker image, where prabath is the Docker ID, which you need to replace with your own:

```
\> docker tag defa4cc5639e prabath/manning-sts-appendix-a
```

Finally, we can push the tagged Docker image to Docker Hub with the following command:

```
\> docker push prabath/manning-sts-appendix-a
```

When we publish prabath/manning-sts-appendix-a image to Docker Hub, we in fact publish it to the Docker Hub registry under prabath/manning-sts-appendix-a repository. Now anyone having access to Docker Hub can pull this Docker image and spin up a container, with the same command we used in section A.4.6, but with the image name prabath/manning-sts-appendix-a.

```
\> docker run -p 8443:8443 prabath/manning-sts-appendix-a
```

A.8 Image name and image id

When we build a Docker image it has to have a name. In section A.7, we used prabath/manning-sts-appendix-a as our image name. To be precise, prabath/manning-sts-appendix-a is not the image name, but the name of the corresponding repository (still many used to called it image name too, which is fine). As we discussed in section A.6, a repository can have multiple versions of a given Docker image. In other words, the registry, the repository and the image version (or the tag) uniquely identify a given Docker image.

The image version is commonly known as a tag. A tag is the same as a version, which we use with other software. If you look at the Tomcat application server image (https://hub.docker.com/_/tomcat?tab=tags) in Docker Hub, you will find multiple tags. Each image tag represents a Tomcat version number. That is a practice we follow. To pull the Tomcat image from Docker Hub, we have two options. One is to pull the image, just by the repository name (without a tag) and the other one is to pull the image both by the repository name and the tag.

A.8.1 Docker images with no tags (or the latest tag)

Let's see how to work with Docker images with no tags. The following docker command pulls a Tomcat image without a tag.

```
\> docker pull tomcat
```

To see the tag of the Tomcat image pulled from Docker Hub, let's use the following docker command.

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|--------------|-------|
| tomcat | latest | 1c721f25f939 | 11 hours ago | 522MB |

The above output says the tag of the Tomcat image is **latest**. This is a special tag in Docker, and also the most confusing tag. If you have published an image to Docker Hub, without a tag (just by the repository name, as we did in section A.7), Docker Hub by default assigns it the **latest** tag. It is just a name – and it does not mean it is the latest Tomcat image in that repository. It is a Tomcat image, where the Tomcat developers have published to Docker Hub without a tag. Also, when you use `docker pull` or `docker run`, just with the repository name of the image you want to have (without the tag), Docker Hub thinks, it is a request for the Docker image with the **latest** tag. That is why earlier, when we use `docker pull tomcat` command, Docker pulled the corresponding Tomcat image with the **latest** tag. Even if we had a more recent Tomcat image with a tag, still Docker will pull the Tomcat image with the **latest** tag, because we skipped the tag in our command. What if all the images in Tomcat repository are tagged? Then, the above command, with no tag, will result in an error.

A.8.2 Docker images with a tag

As a best practice we should tag all the images we push to Docker Hub – at least that will avoid the confusion, with the **latest** tag. Let's try to repeat the steps we followed in section A.7 to publish a Docker image to the Docker Hub, but this time with a tag. We are not going to explain each step here – but only the steps that we need to add a tag. Before we push the image to Docker Hub, in addition to the repository name (`prabath/manning-sts-appendix-a`), we also need to provide a tag (v1 in this case). Then we can use the `docker run` command, along with the image tag – to pull the exact version we pushed to Docker Hub under v1 tag.

```
\> docker tag defa4cc5639e prabath/manning-sts-appendix-a:v1
\> docker push prabath/manning-sts-appendix-a:v1
\> docker run -p 8443:8443 prabath/manning-sts-appendix-a:v1
```

A.8.3 Working with 3rd party Docker registries

Although we mentioned in section A.8 intro, the registry, the repository and the image version (or the tag) uniquely identify a given Docker image, we never used the registry name in any of the commands we used. Probably you might have guessed the reason already! By default Docker uses Docker Hub as the registry – and if we do not explicitly tell Docker to use a 3rd party registry, it will simply use Docker Hub. Let's assume we have our `prabath/manning-sts-appendix-a` repository in Google Container Registry (GCR), and then this is how we use `docker run`, to pull our image from GCR, where `gcr.io` is the endpoint of GCR:

```
\> docker run -p 8443:8443 gcr.io/prabath/manning-sts-appendix-a:v1
```

To publish an image to GCR, we can use the following commands.

```
\> docker tag e7090e36543b gcr.io/prabath/manning-sts-appendix-a:v1
\> docker push gcr.io/prabath/manning-sts-appendix-a:v1
```

A.8.4 Docker Hub official and unofficial images

Docker Hub maintains a set of official Docker images – and the rest is just unofficial images. When you publish an image to Docker Hub, and do nothing, it is just an unofficial image. The repository name of an unofficial Docker image must qualify with the Docker Hub username. That is why we had to use `prabath/manning-sts-appendix-a` as our repository name, instead of just `manning-sts-appendix-a`, where `prabath` is the Docker Hub username.

Publishing an official image to Docker Hub requires more work as defined here: https://docs.docker.com/docker-hub/official_images/. There is a dedicated team from Docker Inc. for reviewing and publishing all the content in an official image. As a user of an official image, the difference we see is how the repository name is constructed. The official repository names do not need to qualify with the Docker Hub username. For example, following command pulls the official Tomcat image from Docker Hub.

```
: \> docker pull tomcat
```

A.8.5 Image id

We can represent each Docker image as a JSON object and this object is written in to a JSON file, inside the Docker image. Let's have a look at it. We can use the following command to pull the Tomcat Docker image having the tag `9.0.20` from Docker Hub and save in to a file called `tomcat.9.0.20.tar`.

```
\> docker save -o tomcat.9.0.20.tar tomcat:9.0.20
```

The above command will create the `tomcat.9.0.20.tar` file with all the image content. It's in the tar, compressed format – and we can use some tar utility (based on your operating system) to decompress it.

```
\> tar -xvf tomcat.9.0.20.tar
```

This will produce a set of directories with long names – and a JSON file. For our discussion here what matters is this JSON file and it is the representation of the `tomcat:9.0.20` Docker image. You can use some tool, based on your operating system to calculate the SHA256 digest of this file. SHA256 is a hashing algorithm, which creates a fixed length (256bits) digest from any given content. Here we are using OpenSSL to generate the SHA256 of the JSON file.

```
\> openssl dgst -sha256 e9aca7f29a6039569f39a88a9decdfb2a9e6bc81bca6e80f3e21d1e0e559d8c4.json
SHA256(e9aca7f29a6039569f39a88a9decdfb2a9e6bc81bca6e80f3e21d1e0e559d8c4.json)=
e9aca7f29a6039569f39a88a9decdfb2a9e6bc81bca6e80f3e21d1e0e559d8c4
```

Looking at the output of the above command, you might have already noticed some pattern. Yes! It is not a coincident. The name of the file is in fact the SHA256 hash of that file – that is how this file is named. Now the mystery is over! The image id of a Docker image is the SHA256 hash of the corresponding JSON file – to be precise; it is in fact the hexadecimal representation of the SHA256 hash. Let's use the following docker command to clarify it. It prints the image id of the given Docker image, which is the 1st 12 digits of the image id we figured out before.

```
\> docker images tomcat:9.0.20
REPOSITORY      TAG          IMAGE ID            CREATED        SIZE
tomcat          9.0.20      e9aca7f29a60    14 hours ago   639MB
```

A.8.6 Pulling an image with the image id

To pull an image from a Docker registry, so far we used the image name, along with the tag. For example, the following command pulls the Tomcat image with the tag 9.0.20.

```
\> docker pull tomcat:9.0.20
```

There are few things we need to worry about, when we pull an image by the tag. Tags are not immutable – means they are subject to change. Also, we can push multiple images with the same tag to a Docker registry – and the latest will override the previous image with the same tag. So, when we pull an image by tag, there is a chance we may not get the same image all the time. Pulling an image with the image id helps overcoming that issue. The image id is the SHA256 hash of the image. If the content of the image changes, the hash of the image changes as well – hence the image id. There can never be two different images carrying the same image id. So, when we pull an image by the id (or the hash of the image), we always get the same image. Following command shows how to pull an image by the image id. The text after `tomcat@` in the command is the image id of the Tomcat image we want to pull.

```
\> docker pull tomcat@sha256:b3e124c6c07d761386901b4203a0db2217a8f2c0675958f744bbc85587d1e715
```

A.9 Image layers

The Docker image we built in section A.4.4 and published to the Docker Hub in section A.7 has six layers. When we run `docker build` command to build a Docker image, Docker creates a layer for each instruction in the corresponding Dockerfile, except for few instructions such as ENV, EXPOSE, and CMD. For clarity, we repeat the same Dockerfile used in section A.4.4 in listing A.2.

Listing A.2. The content of the Dockerfile

```
FROM openjdk:8-jdk-alpine
ADD target/com.manning.mss.appendixa.sample01-1.0.0.jar /com.manning.mss.
appendixa.sample01-1.0.0.jar
ADD keystores/keystore.jks /opt/keystore.jks
ADD keystores/jwt.jks /opt/jwt.jks
ENTRYPOINT ["java", "-jar", "com.manning.mss.ch10.sample01-1.0.0.jar"]
```

Also when we pull a Docker image from Docker Hub, Docker pulls the image in layers. Following command inspects `prabath/manning-sts-appendix-a` image and in the truncated output shows the SHA256 has of the associated layers.

```
\> docker inspect prabath/manning-sts-appendix-a
"Layers": [
  "sha256:f1b5933fe4b5f49bbe8258745cf396afe07e625bdab3168e364daf7c956b6b81",
  "sha256:9b9b7f3d56a01e3d9076874990c62e7a516cc4032f784f421574d06b18ef9aa4",
  "sha256:ceaf9e1ebef5f9ea707a838848a3c13800fcf32d7757be10d4b08fb85f1bc8a",
  "sha256:52b4aac6cb4680220c70a68667c034836839d36d37f3f4695d129c9919da9e3a",
```

```
"sha256:1fa9ff776ce74439b3437cdd53333c9e2552753cf74986e1f49ca305ad2e3c02",
"sha256:93fc1841ffce641e1da3f3bda10416efcbff73dd9bfe2ad8391683034942dbd5"
]
```

Each layer in a Docker image is read-only and has a unique identifier. Each of these layers is stacked over the other. When we create a container from an image, Docker adds another layer, which is read/write, on top of all the read-only layers. This is called the container layer. Any writes from the container, while it is running are written to this layer. Since the containers are immutable, any data written to this layer will vanish after you remove it. In section A.12 we discuss an approach to make the container's runtime data persistence, specially the logs.

The layered approach Docker follows while creating an image promotes reusability. When you run multiple containers from the same image, each container shares the image (the read-only layers), but has its own independent container layer on top of that. Also even between different images, if they depend on the same image layer, Docker reuses such layers.

A.10 Container lifecycle

A container is an instance of an image. Once we create a container, it can go through multiple phases in its lifecycle: created, running, paused, stopped, killed and removed (figure A.7).

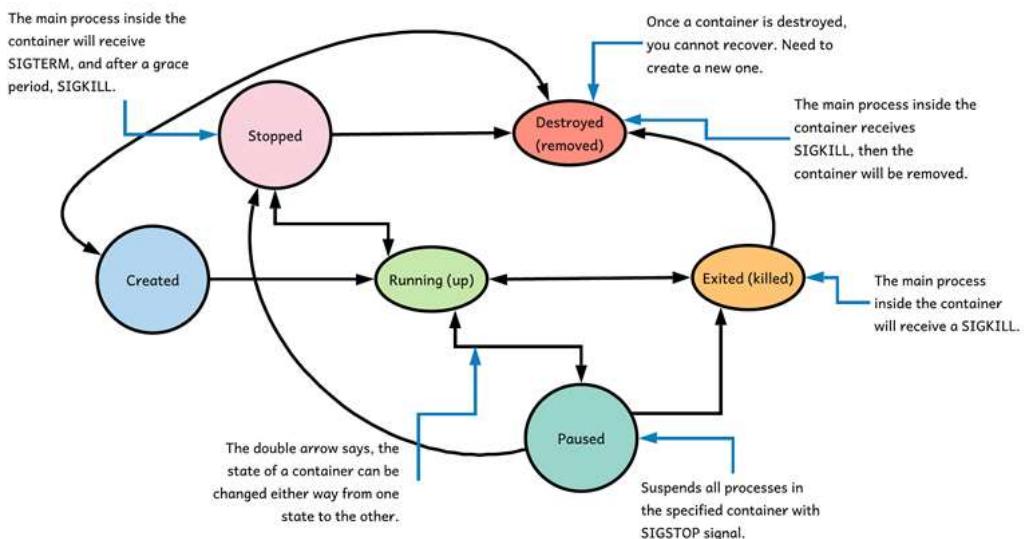


Figure A.7 The status of a container changes from created to destroyed in different phases.

A.10.1 Create a container from an image

Earlier in this appendix we used `docker run` command to start a container. It does two things in a single command. First it creates a container and then starts it. Instead of that we can use

`docker create`, only to create a container from a given Docker image. Following command pulls the Tomcat Docker image from Docker Hub and creates a container with the name `my-web-server`. Once the container is created successfully, the command prints the generated container id.

```
\> docker create --name my-web-server tomcat
d9ad318893e3844695d9f25dd53cb88128329b90d024c86d1774475b81e39de3
```

We can use the following command to view the status of the latest (-l) created container. Here you can see `my-web-server` container carries the `Created` status.

```
\> docker ps -l
CONTAINER ID      IMAGE       STATUS        NAMES
d9ad318893e3      tomcat      Created       my-web-server
```

What does it mean that the container is created, but not running or up? The `docker create` command takes the corresponding image – and on top of that adds a new writable layer and prepares it for running.

A.10.2 Start a container

We can start a Docker container, which is in the created or stopped (not paused) status with the `docker start` command.

```
\> docker start my-web-server
```

Once the container is booted up, the following command shows its status as up.

```
\> docker ps -l
CONTAINER ID      IMAGE       STATUS        NAMES
d9ad318893e3      tomcat      Up 7 seconds  my-web-server
```

A.10.3 Pause a running container

We can use `docker pause` command to suspend a running container. This sends a **SIGSTOP**² signal to all the processes running in the corresponding container. What does this really mean? If we have some data written to the containers writable layer, would that data be still available when we un-pause the container to bring it back to the running status (section A.10.4)? Yes, pausing a container will not remove its writable layer and the data already written will stay unchanged. Docker cleans up the data written to the container layer, only when we remove a container from the system (section A.10.6) – even killing a container (section A.10.5) will not wipe out the container layer.

```
\> docker pause my-web-server
```

Once the container is paused, the following command shows its status as paused.

```
\> docker ps -l
```

² If a process receives a **SIGSTOP** it is paused by the operating system. All its state is preserved ready for it to be restarted but it doesn't get any more CPU cycles until then.

| CONTAINER ID | IMAGE | STATUS | NAMES |
|--------------|--------|----------------------------|---------------|
| d9ad318893e3 | tomcat | Up About a minute (Paused) | my-web-server |

We can use the following command to bring back a paused container to the running status.

```
\> docker unpause my-web-server
```

Just as the container layer of a paused container remains unchanged after we un-pause it or bring it back to the running status, any data you write to the container's memory will also remain unchanged. This is because the `docker pause` does not make the processes running in the container to restart when we un-pause the container.

A.10.4 Stop a running container

Once a container is in running state, we can use `docker stop` to stop it. This sends the main process inside the container SIGTERM signal, and after a grace period, the SIGKILL signal. Once an application receives a SIGTERM signal it can determine what it needs to do. Ideally it will clean up any resources and then stop. For example, if your microservice running in a container already has some inflight requests, which it is currently serving, then it can work on those, while not accepting any new requests. Once all the inflight requests are served, the service can stop itself. In case, the service decides not to stop it by itself, the SIGKILL signal generated after some grace period will make sure, that the container is stopped no matter what.

```
\> docker stop my-web-server
```

Once the container is stopped, the following command shows its status as exited with status code 143.

| CONTAINER ID | IMAGE | STATUS | NAMES |
|--------------|--------|----------------------------|---------------|
| d9ad318893e3 | tomcat | Exited (143) 3 seconds ago | my-web-server |

We can use the following command to bring back an exited container to the running status.

```
\> docker restart my-web-server
```

Even though the container layer of the stopped container remains unchanged after we restart it, any data you write to the container's memory will be lost (unlike in `docker pause`). This is because the `docker stop` makes the main process running in the container to stop and when we restart it, the process restarts too.

A.10.5 Kill a container

We can kill a container, which is running or in the paused status, using `docker kill`. This sends the main process inside the container SIGKILL signal, which will immediately take down the running container. To stop a container, we should always use `docker stop`, instead of `docker kill`, as the `stop` command gives control to the running process to do some cleanup.

```
\> docker kill my-web-server
```

Once the container is killed, the following command shows its status as exited with status code 137.

```
\> docker ps -l
CONTAINER ID        IMAGE       STATUS        NAMES
d9ad318893e3      tomcat     Exited (137) 20 seconds ago   my-web-server
```

We can use the following command to bring back an exited container to the running status. Even though the container layer of the killed container remains unchanged after we restart it, any data you write to the container's memory will be lost. This is because the `docker kill` makes the main process running in the container to stop and when we restart it, the process restarts too.

```
\> docker restart my-web-server
```

Even though the container layer of the killed container remains unchanged after we restart it, any data you write to the container's memory will be lost (like in `docker stop`). This is because the `docker kill` makes the main process running in the container to stop and when we restart it, the process restarts too.

A.10.6 Destroy a container

We can use `docker rm` to remove a container from Docker host. We have to stop the running container first, either using `docker stop` or `docker kill`, before removing it.

```
\> docker rm my-web-server
```

There is no way to bring back a removed container – we have to create it again.

A.11 Deleting an image

When we pull an image from a Docker registry (or create our own as in A.4.4), it is stored in Docker host. We can use the following command to remove the copy of the Tomcat image with the tag 9.0.20 stored in Docker host.

```
\> docker image rm tomcat:9.0.20
```

While deleting an image, Docker deletes all the image layers, unless other images in the system have any dependencies to those. In case there is a container in the system, already using the image we want to delete, Docker will not allow. First we need to remove the container using `docker rm` command.

A.12 Docker volumes

Containers are immutable. Any data written into the container file system (or the container layer) vanishes soon after we remove it. But still we cannot let that happen. Immutable containers are important in a containerized environment, so that we can spin up and spin down

containers from a Docker image and bring them into the same state. But still there are some data we need to persist, for example runtime logs. We cannot afford to lose logs.

Docker volumes help persisting runtime data of a container. For example, we can have one container writing logs to a volume, and another container sharing the same volume can read the logs and possibly push the logs to a log management system like Fluentd. In the same way, if you want to store log data in Amazon S3, then you can have another container to read the logs from the shared volume and publish them to S3. The container, which writes logs to the volume, does not necessarily need to know where to publish them or how to publish them, but a different container can take up that responsibility. This is a good way of implementing separation of concerns.

Let's take a simple example to see how volumes work. Here we are going to spin up a Tomcat container and save its log files in a directory in the host machine. So, even we take the container down, we still have the logs. Let's use the following command to spin up a Tomcat container with a volume mapping. The `-v` argument in the `docker run` command says, map `/usr/local/tomcat/logs` directory from the container file system to `~/tomcat/logs` directory in our host file system. If you do not have a directory called `~/tomcat/logs` in your host file system, you need to create it first before running `docker run`.

```
\$ docker run --name tomcat -v ~/tomcat/logs:/usr/local/tomcat/logs tomcat:9.0.20
```

Once the container starts up, we can find all the logs inside the volume we created in the host file system under, `~/tomcat/logs` directory. Even after we take down the container, the log files will remain in the host file system.

A.13 Docker internal architecture

Docker initially used Linux containers (LXC) to bring process isolation. Linux cgroups and namespaces are the two fundamental technologies behind LXC. The cgroups and namespaces are implemented at Linux kernel, but not LXC. When Docker was first released in 2013, it included the Docker daemon, where the implementation was based on LXC. The use of LXC as the execution environment for Docker dropped from version 1.10 onward, after making it optional since 0.9. Libcontainer, which was developed by Docker Inc. replaced LXC. Libcontainer is the default execution environment of Docker now. The motivation behind building libcontainer was to get direct access to the kernel level constructs to build and run containers, rather than going through LXC (as LXC is not part of the Linux kernel). Libcontainer is a library, which interacts with cgroups and namespaces at the Linux kernel level. In addition to dropping support for LXC, Docker also worked on breaking down the monolithic Docker daemon and taking some of functionalities out of it, which was also the motivation behind the Moby project, which we discuss under the section A.19. Figure A.8 shows the Docker internal architecture.

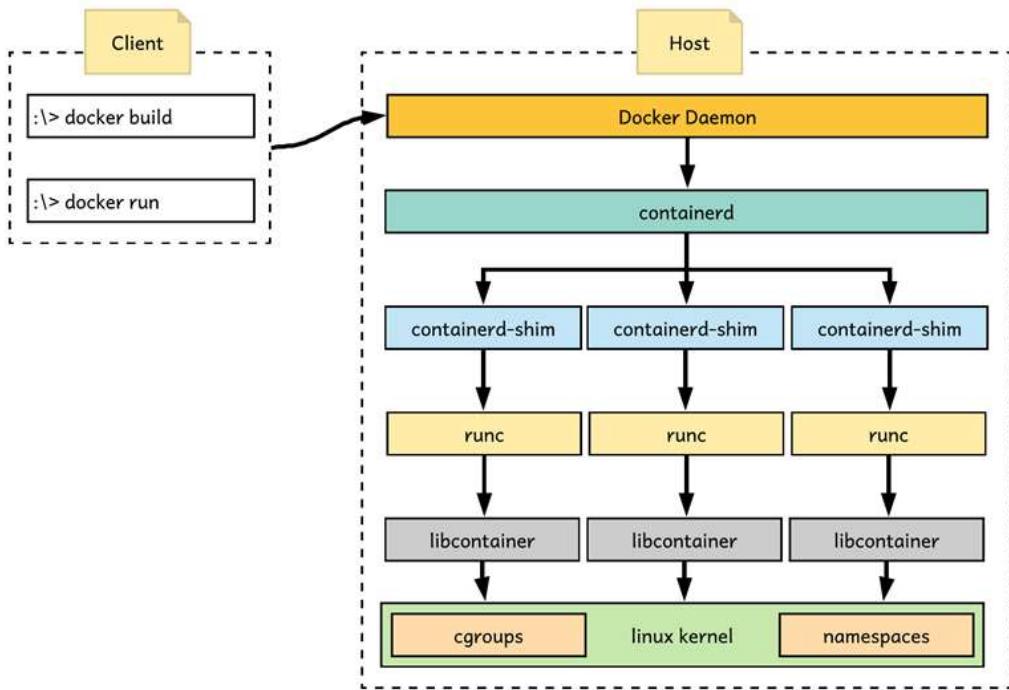


Figure A.8 Docker internal architecture. cgroups and namespaces, which are implemented at Linux kernel, are the fundamental building blocks of Docker containers.

A.13.1 Containerd

Containerd is the component in the Docker internal architecture that is responsible for managing containers across all the lifecycle stages. It takes care of starting, stopping, pausing, and destroying containers. Containerd finds the corresponding Docker image, based on the image name that comes along with the API request, which is passed to it by the Docker daemon and converts the image to an Open Container Initiative (OCI) bundle, and then passes the control to another component called containerd-shim. OCI (<https://www.opencontainers.org/>) is a standard body established in June 2015, by Docker and others in the container industry to develop standards around container formats and runtime. Docker supports OCI specifications from version 1.11 onward.

A.13.2 Containerd-Shim

Containerd-shim forks another process called runc, which internally talks to the operating system kernel to create and run containers. Runc starts a container as a child process of it and once the container starts running, runc kills itself.

A.13.3 Runc

Runc implements Open Container Initiative (OCI) container runtime specification. The responsibility of runc is to create containers by talking to the operating system kernel and it uses the libcontainer library to interact with cgroups and namespaces at the kernel level.

A.13.4 Linux namespaces

Docker brings process isolation with namespaces. A namespace in Linux partitions kernel resources, so that each running process will have its own independent view of those resources. While implementing namespaces, Linux introduced two new system calls: *share()* and *setns()*, together with six new constant flags. Each of these new constant flags represents a namespace, as listed below:

- **PID namespace** (identified by CLONE_NEWPID flag): Ideally when we run a process on the host operating system, it gets a unique process identifier. When we have a partitioned process ID (PID) namespace, each container can have its own process identifiers independent of the other processes running on other containers (on the same host machine). By default each container has its own PID namespace, but if we want to share a PID namespace between multiple containers, we can override the default behavior by passing the `--pid` argument to `docker run` command. The following command forces the hello-world Docker container to use the PID namespace of the host machine:

```
: \> docker run --pid= "host" hello-world
```

The following command forces the hello-world Docker container to use the PID namespace of the container `foo`:

```
: \> docker run --pid= "container:foo" hello-world
```

- **UTS namespace** (identified by CLONE_NEWUTS flag): Unix Time Sharing (UTS) namespace isolates the hostname and the Network Information Service (NIS) domain name. In other words, the UTS namespace isolates hostnames and each container can have its own irrespective of what hostnames other containers have. To override this default behavior to share the UTS namespace with the host machine, we can pass the `-uts` argument with value `host` to the `docker run` command:

```
: \> docker run --uts= "host" hello-world
```

- **NET namespace** (identified by CLONE_NEWWNET flag): The NET namespace isolates the network stack with all the routes, firewall rules and the network devices. For example, two processes running in two different containers can listen on the same port, with no conflict. We discuss Docker networking in detail under the section A.9.
- **MNT namespace** (identified by CLONE_NEWNS flag): The MNT (mount) namespace isolates the mount points in the system. In simple terms, a mount point defines where your data is. For example, when you plug in a USB pen drive in to your MacBook, it auto

mounts the pen drive's file-system to `/Volumes` directory. The mount namespace helps isolating one container's view of the file system from other containers', as well as the host file system. Each container will see its own `/usr`, `/var`, `/home`, `/opt`, `/dev` directories.

- **IPC namespace** (identified by `CLONE_NEWIPC` flag): The IPC namespace isolates the resources related to inter-process communication: memory segments, semaphores and message queues. By default each container has its own private namespace and we can override that behavior by passing `--ipc` argument to the `docker run` command. The following command forces the hello-world Docker container to join the IPC namespace of another container (`foo`):

```
: \> docker run --ipc= "container:foo" hello-world
```

- **USR namespace** (identified by `CLONE_NEWUSER` flag): The USR namespace isolates the user identifiers within a given container. This allows two different containers to have two users (or groups) with the same identifier. Also you can run a process as a root in a container, while (the same process) having no root access outside the container.

A.13.5 Linux cgroups

Control groups (cgroups), is a Linux kernel feature that allows controlling resources allocated to each process. With cgroups we can say how much of CPU time, how many CPU cores, how much of memory, and so on a given process (or a container) is allowed to use. This is extremely important in an environment where the same set of physical resources (CPU, memory, network, etc) from the host machine is shared among multiple containers to avoid one container from misusing them. We can restrict the resource usage of a given container by passing the amount of resources allocated to it as arguments to the `docker run` command. The following command sets the maximum memory usage to 256 MB to the container starts from the hello-world Docker image:

```
\> docker run -m 256m hello-world
```

The following command allows the container to use CPU core 1 or CPU core 2:

```
\> docker run --cpuset-cpus="1,2" hello-world
```

A.14 What is happening behind the scenes of docker run?

When we execute the command `docker run` via a Docker client, it talks to an API running on Docker daemon (see figure A.9). Docker client and daemon can run on the same host machine or multiple machines. Once the Docker **daemon** receives the API request to create and spin up a new container, it internally talks to another component called **containerd**, which we discussed earlier in this chapter. Containerd finds the corresponding Docker image, based on the image name comes along with the API request and converts it to an Open Container Initiative (OCI) bundle, and then passes the control to another component called, `containerd-shim`. **Containerd-shim** forks another process called **runc**, which internally talks to the

operating system kernel to create and run the container. Runc starts the container as a child process of it and once the container starts running, runc kills itself. The figure A.9 illustrates the flow of spinning up a container.

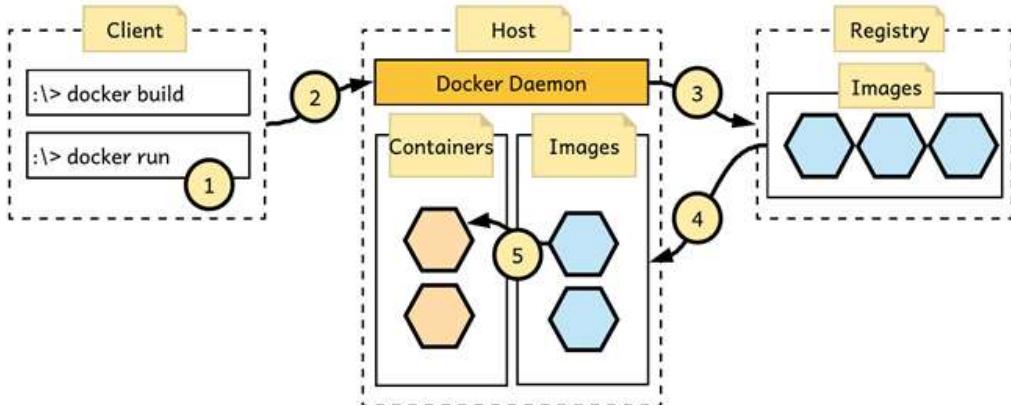


Figure A.9 Behind the scenes of docker run command.

Under section A.3, we used the following command to run the hello-world process in a container:

```
\>docker run hello-world
```

As you might have already noticed, when we run the container as in the above command, the console or the terminal we run the command, automatically gets attached to the container's standard input, output and error. This way of running a container is known as foreground mode and it is the default behavior. If we want to start the container in the detached mode, then we need to pass the `-d` argument to the `docker run` command and in the output it will print the container id.

```
\>docker run -d hello-world
74dac4e3398410c5f89d01b544aaee1612bb976f4a10b5293017317630f3a47a
```

Once the container got started, we can use the following command to connect to the container with the container id, and get the logs:

```
\> docker container logs 74dac4e3398410c5f89d01b544aaee1612bb...
```

Following is another useful command to inspect the status and properties of a running container:

```
\> docker container inspect 74dac4e3398410c5f89d01b544aaee1612bb...
[
    {
        "Id": "74dac4e3398410c5f89d01b544aaee1612bb976f4",
        ...
    }
]
```

```

    "Created": "2019-05-22T18:29:47.992128412Z",
    "Path": "/hello",
    "Args": [],
    "State": {
        "Status": "exited",
        "Running": false,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 0,
        "ExitCode": 0,
        "Error": "",
        "StartedAt": "2019-05-22T18:29:48.609133534Z",
        "FinishedAt": "2019-05-22T18:29:48.70655394Z"
    },
}
]

```

A.15 Inspecting traffic between Docker client and host

All the Docker commands you have tried out so far are generated from your Docker client and sent over to the Docker host. Let's try to inspect the traffic between the Docker client and the host, using a tool called socat. This will help you understand, what is happening underneath. First you need to install socat on your operating system. If you are on Mac, you can use brew, or if you are on Debian/Ubuntu Linux, you can use apt-get. The best way to find the installation steps is to Google for socat with your operating system name. We could not find a single place, which carries installation instructions for all the operating systems. Once you have socat installed, you can use the following command to run it.

```
\> socat -d -d -v TCP-L:2345,fork,bind=127.0.0.1 UNIX:/var/run/docker.sock
2019/06/24 23:37:32 socat[36309] N listening on LEN=16 AF=2 127.0.0.1:2345
```

The `-d -d` flags in the above command asks socat to print all fatal, error, warning and notice messages. If you add another `-d` it will also print info messages. According to the manual (man page) of socat, the `-v` flag instructs socat to write the transferred data not only to their target streams, but also to stderr. The `TCP-L:2375` flag instructs socat to listen on port 2375 for TCP traffic. The `fork` flag enables socat to handle each arriving packet by its own sub process. In other words, when we use `fork`, socat creates a new process for each newly accepted connection. . The `bind=127.0.0.1` flag instructs socat to listen only on the loopback interface, so no one outside the host machine can directly talk to socat. The `UNIX:/var/run/docker.sock` is the address of the network socket, where the Docker daemon accepts connections. In effect, the above command asks socat to listen for TCP traffic on port 2345, log them and forward them to the Unix socket `/var/run/docker.sock`.

By default, the Docker client is configured to talk to a Unix socket. But to intercept traffic between the client and the Docker host, we need to instruct Docker client to send requests via socat. You can run the following command to override the `DOCKER_HOST` environment variable and point it to socat.

```
\> export DOCKER_HOST=localhost:2345
```

Let's run the following Docker command (from the same terminal you exported the DOCKER_HOST environment variable), for example, to find all the Docker images available in the Docker host.

```
\> docker images
```

While running the above command, also observe the terminal, which runs socat. There you will find, request and response messages between Docker client and host are printed. Without going through the `docker images` command, you can get the same results from the following curl command (assuming Docker daemon and socat are running on localhost), by talking to the Docker API.

```
\> curl http://localhost:2345/v1.39/images/json
```

A.16 Docker compose

Docker compose is an open source tool (not part of the Docker engine) written in Python, that helps you manage multi-container applications. For example, your application may have a microservice, a database and a message queue. Each of these components runs in its own container. Rather than managing those containers independently, we can create a single yaml file called, `docker-compose.yaml` – and define all the required parameters and dependencies there. To start the application with all three containers, we only need to run a single command: `docker-compose up` from the directory where you have the `docker-compose.yaml` file. By default, `docker-compose` looks for a file with the name `docker-compose.yaml` in its current directory, but you can override the default behavior with the `-f` argument and pass the name of the yaml file (`docker-compose -f my-docker-compose.yml up`). Listing A.3 shows a sample, `docker-compose.yaml` file.

Listing A.3. The content of the docker-compose.yaml file

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper
  kafka:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  depends_on:
    - "zookeeper"
  sts:
    image: prabath/manning-sts-appendix-a
    ports:
      - "8443:8443"
  depends_on:
    - "kafka"
```

The above docker-compose.yaml, defines three containers for the application. The security token service (STS) is a microservice, which issues tokens – and it depends on Kafka, which is a message broker. Kafka internally depends on ZooKeeper for distributed configuration management. Effectively to run our STS application we need three Docker images. We can start all three containers with the single `docker-compose up` command. A detailed discussion on Docker compose is out of the scope of this book and if you are still keen on understanding Docker compose in detail, please refer online documentation: <https://docs.docker.com/compose/overview/> or the chapter 11 of the book, Docker in Action (Manning Publications, 2019) by Jeff Nickoloff and Stephen Kuenzli.

A.17 Docker Swarm

In practice, when we run Docker in production, we may have hundreds of Docker containers running in multiple nodes. A node can be a physical machine or a virtual machine. Also, thinking in terms of high-availability of the application we run in a container, we need to have multiple replicas of the same application running in different containers. To decide the number of replicas for a given application, we need to know how much load (traffic) one single container can handle – and also the average and the peak load our system gets. It is desirable that we run the minimal number replicas to handle the average load – and auto scale (spin up more replicas) as the load increases. With that approach we waste a minimal amount of system resources.

Docker swarm addresses most of the above concerns and more. It is a feature, which is part of the Docker engine since Docker 1.12 – prior to that it was an independent product. There are two main objectives of Docker swarm, one is to manage a multi-node Docker cluster and the other is to act as an orchestration engine for Docker containers. These features are built into Docker with the open source project called, SwarmKit (<https://github.com/docker/swarmkit/>).

Let's revisit the high-level Docker architecture we discussed under section A.3. As figure A.9 shows, it only worries about a single Docker node. If we want to run a container, the Docker client over an API talks directly to the Docker host to run the container. In a Docker cluster with multiple nodes, it is impractical for our Docker client to talk to multiple Docker hosts and schedule to run a container. This is one very basic limitation – and there are many to address the requirements we discussed at the beginning of this section – and the architecture presented in figure A.10, no longer going to work in a Docker cluster.

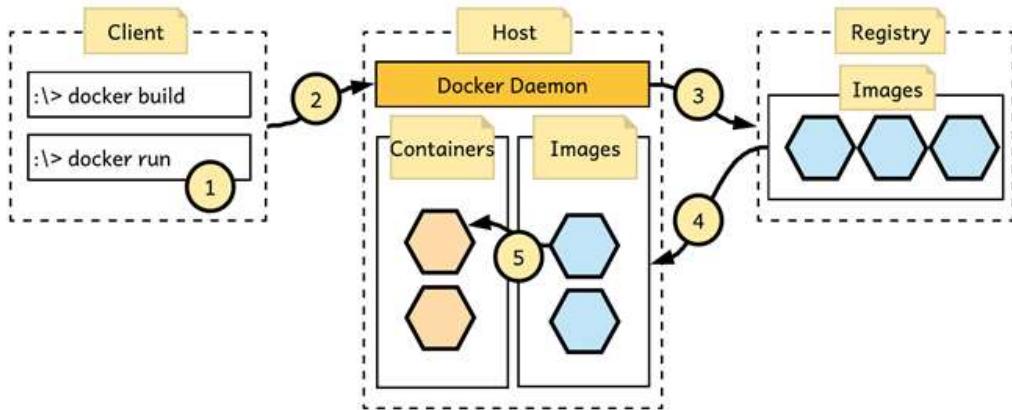


Figure A.10 High-level Docker component architecture. The Docker client talks to the Docker daemon running on the Docker host over a REST API to perform various operations on Docker images and containers.

The new architecture Docker swarm proposes (figure A.11), introduces a set of manager nodes and a set of worker nodes. In other words, a Docker cluster is a collection of manager nodes and worker nodes. The manager nodes in a cluster are responsible for managing the state of the cluster. For example, if our requirement is to run 5 replicas of a given application, then the manager nodes must make sure it happens. Also, if our requirement is to spin up more replicas as the load goes up and spin down as the load goes down, the manager nodes should generate the control signals to handle that situation, by monitoring the load on that particular application. The responsibility of the worker nodes is to accept control signals from manager nodes and act accordingly. In fact, containers run on worker nodes. To schedule an application to run on the Docker cluster, the Docker client talks to one of the manager nodes. The set of manager nodes in a cluster is collectively known as the control plane.

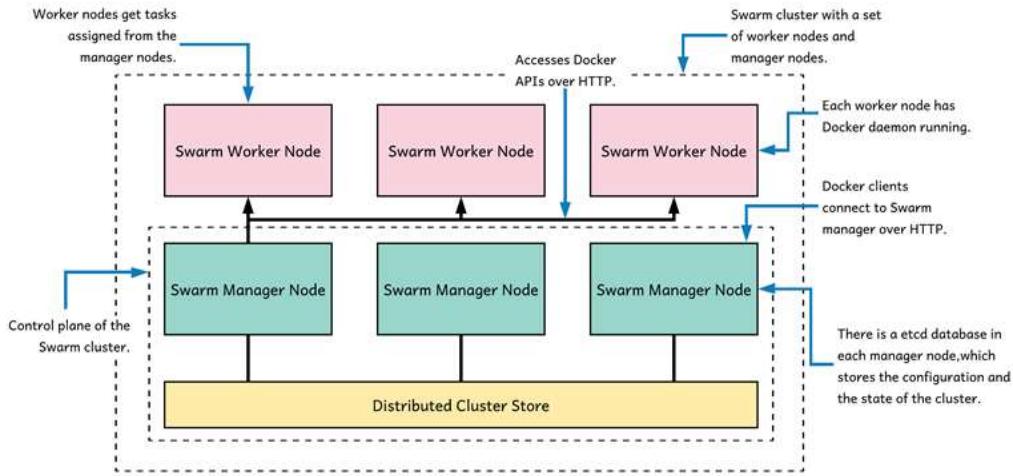


Figure A.11 High-level Docker Swarm architecture. A Swarm cluster is built with a set of manager nodes and a set of worker nodes.

Swarm introduces a new concept called – a service. A service is the smallest deployment unit a Docker cluster – not a container. In fact, a service builds a wrapper over a Docker container. A container wrapped in a service is also known as a replica or a task. We use `docker service create` command to create a Docker service. In addition to the image name, port mapping and other arguments we use in `docker run` command, the `docker service create` command also accepts number of other parameters, which you check with the command, `docker service create -help`. The `replicas` argument in the following command asks swarm to create 5 replicas of the corresponding service (or the application).

```
\$ docker service create --name hello-service --replicas 5 hello-world
```

An in depth walk-through of swarm is out of the scope of this book, and if you are interested in reading more, we would recommend checking out the chapter 12 of the book *Docker in Action* (Manning Publications, 2019) by Jeff Nickoloff and Stephen Kuenzli. Also the online documentation on Docker swarm is a very good place to start with: <https://docs.docker.com/engine/swarm/key-concepts/>.

A.18 Docker networking

As discussed in section A.13.4 the NET namespace in the Linux kernel provides an isolated networking stack with all the routes, firewall rules and the network devices for a container. In other words, a Docker container has its own networking stack. Docker networking is implemented with the libnetwork open source library written in Go programming language, based on the Container Network Model (CNM) design specification. CNM defines core building blocks for Docker networking (figure A.12): sandbox, endpoint and network.

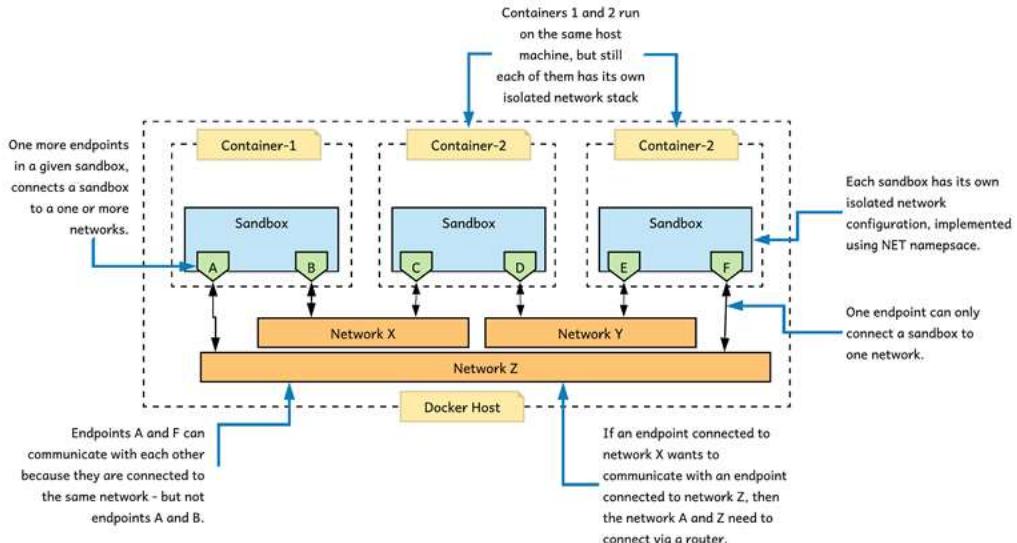


Figure A.12 Docker follows Container Network Model (CNM) design for networking, which defines three building blocks: sandbox, endpoint and network.

A sandbox is an abstraction over an isolated networking configuration. For example, each container has its own network sandbox, with routing tables, firewall rules and Domain Name Services (DNS) configuration. Docker implements a sandbox with the NET namespace. An endpoint represents a network interface and each sandbox has its own set of endpoints. These virtual network interfaces connect a sandbox to the network. A set of endpoints that needs to communicate with each other forms a network. By default Docker supports three networking modes: bridge, host and none. The following command lists out the supported Docker networks:

```
:> docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
f8c9f194e5b7    bridge    bridge      local
a2b417db8f94    host      host       local
583a0756310a    none     null       local
```

Following command in listing A.4 uses the network ID to get further details of a given network:

Listing A.4. The details of a bridge network

```
\> docker network inspect f8c9f194e5b7
[
  {
    "Name": "bridge",
    "Id": "f8c9f194e5b70c305b3eb938600f9caa8f5ed11439bc313f7245f76e0769ebf6",
    "Created": "2019-02-26T00:20:44.364736531Z",
    "Scope": "local",
```

```

    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": [
        {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        {
            "Internal": false,
            "Attachable": false,
            "Ingress": false,
            "ConfigFrom": {
                "Network": ""
            },
            "ConfigOnly": false,
            "Containers": {},
            "Options": {
                "com.docker.network.bridge.default_bridge": "true",
                "com.docker.network.bridge.enable_icc": "true",
                "com.docker.network.bridge.enable_ip_masquerade": "true",
                "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
                "com.docker.network.bridge.name": "docker0",
                "com.docker.network.driver.mtu": "1500"
            },
            "Labels": {}
        }
    ]
]

```

A.18.1 Bridge networking

The bridge networking in Docker uses Linux bridging and iptables to build connectivity between containers running on the same host machine. It is the default networking mode in Docker. When Docker spins up a new container, it is attached to a private IP address. If you already have hello-world container running from section A.3, following two commands help to find the private IP address attached to it. The first command finds the container ID corresponding to hello-world and the second command uses it to inspect the container. The output of the second command is truncated only to show the network configuration:

```

\> docker ps
CONTAINER ID IMAGE STATUS PORTS
b410162d213e hello-world Up About a minute 0.0.0.0:8443->8443/tcp

\> docker inspect b410162d213e
[
{
    {
        "Networks": {
            "bridge": {
                "Gateway": "172.17.0.1",
                "IPAddress": "172.17.0.2",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": ""
            }
        }
    }
]

```

```

        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02"
    }
}
]

```

The private IP address assigned to a container is not accessible directly from the host machine. When one container talks to another container in the same host machine with the private address, the connection is routed through the docker0 bridge-networking interface, as shown in figure A.13. Anyway for the communication between containers in the same host machine, IP address is not the best option. These IP addresses can change dynamically when containers spins up and down. Instead of the private address, we can also use the container name it self to communicate between containers. This is one benefit of giving a container a meaningful name (rather than relying on a randomly generated UUID by Docker) at the time we spin up a container.

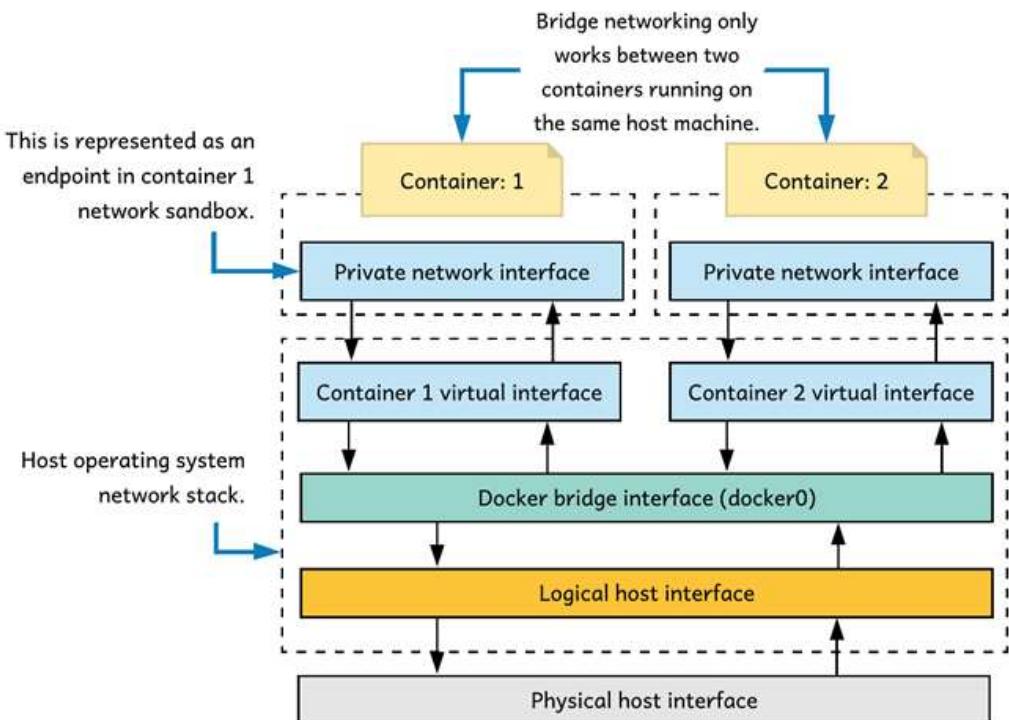


Figure A.13 Containers use the bridge-networking interface provided by Docker to communicate with each other.

A.18.2 Host networking

Host networking is the laziest option in Docker to facilitate communication between containers. When host networking is enabled for a given container, it uses host machine's networking stack directly and also shares the network namespace with the host machine. To enable host networking, we need to pass the `--network` argument to the `docker run` command with the value `host`:

```
\> docker run --network= "host" hello-world
```

A.18.3 No networking

No networking mode disables all the networking interfaces available for a given container. It's a closed container. One would need a container with no networking to carry out some specific tasks, for example to process a set of log files in one format and output into another format. To disable networking, we need to pass the `--network` argument to the `docker run` command with the value `none`:

```
\> docker run --network= "none" hello-world
```

A.18.4 Networking in a Docker production deployment

Bridge networking only works in a single host environment or in other words only between the containers deployed on the same host machine. In a production deployment this is not sufficient and containers have to interact with each other running on different host machines. Container orchestration frameworks like Kubernetes and Docker Swarm support multi-host networking in Docker. Docker Swarm supports multi-host container communication with Docker overlay networking and in chapter 11 we discuss in detail how Kubernetes supports multi-host networking.

A.19 Moby project

Docker (the company) announced Moby project during their annual user conference, DockerCon in 2017. With this announcement, the Docker github project moved from github.com/docker/docker to github.com/moby/moby. Moby project aims to expand the Docker ecosystem by breaking the old monolithic Docker project in to multiple components. Developers from various other projects can reuse these different components to build their own container-based systems. The Moby project has three main components:

- A set of libraries, to address different aspects in a containerized environment. For example, a library to handle networking, another library to handle image management and so on.
- A framework for assembling the components into a standalone container platform, and tooling to build, test and deploy artifacts for these assemblies.
- A reference assembly built using the framework and a set of libraries. This reference assembly is known as Moby Origin, which is the open base for the Docker container platform.

You can read more on the Moby project and its motivation from here:
<https://blog.docker.com/2017/04/introducing-the-moby-project/>

B

Kubernetes fundamentals

Kubernetes¹ is the most popular container orchestration framework. A container is an abstraction over the physical machine, while the container orchestration framework is an abstraction over the network. Container orchestration software like Kubernetes lets you deploy, manage and scale containers in a highly distributed environment with thousands of nodes or even more. Kubernetes has its roots at Google. It started at Google as an internal project, under the name Borg. Borg helped Google developers and system administrators to manage thousands of applications across large datacenters in multiple geographies. Borg became Kubernetes in 2014, which is one of the most popular open source projects today. A detailed discussion on Kubernetes is out of scope of this book and for any readers interested in learning more, we would recommend reading the book *Kubernetes in Action* (Manning Publications, 2017) by Marko Lukša. Also, the book *Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications* (O'Reilly Media, 2019) by Bilgin Ibryam and Roland Huß is a very good reference to learn how to use Kubernetes in a production deployment. In chapter 11 we discuss how to deploy and secure microservices in a Kubernetes environment and if you are new to Kubernetes this appendix lays the right foundation you need in following chapter 11.

B.1 Kubernetes high-level architecture

Kubernetes follows client-server based architecture. A Kubernetes cluster consists of one or more master nodes and one or more worker nodes (see figure B.1). When we want to deploy a microservice in a Kubernetes environment, we directly interact with a Kubernetes master node, which is also known as the Kubernetes control plane. To connect to the Kubernetes master node we have to run a Kubernetes client in our local machine – that is the 3rd component in a Kubernetes environment, in addition to the master nodes and the worker nodes.

¹ The word Kubernetes is little lengthy with 10 characters. In shorter form we call it K8S – where there are 8 characters in between K and S.

B.1.1 Master node

The master node in a Kubernetes cluster, which is also known as the control plane, takes care of almost all the functions (which we discuss later on in this appendix) in a Kubernetes cluster. A Kubernetes master node consists of four main components: API server, controller manager, scheduler, and etcd (see figure B.1). The communication between all the components in a Kubernetes happens via the API server. For example, to deploy a container in Kubernetes, we need to talk to the API server via a Kubernetes client application, which is called kubectl. We discuss all four components in the master node, in detail later in the appendix.

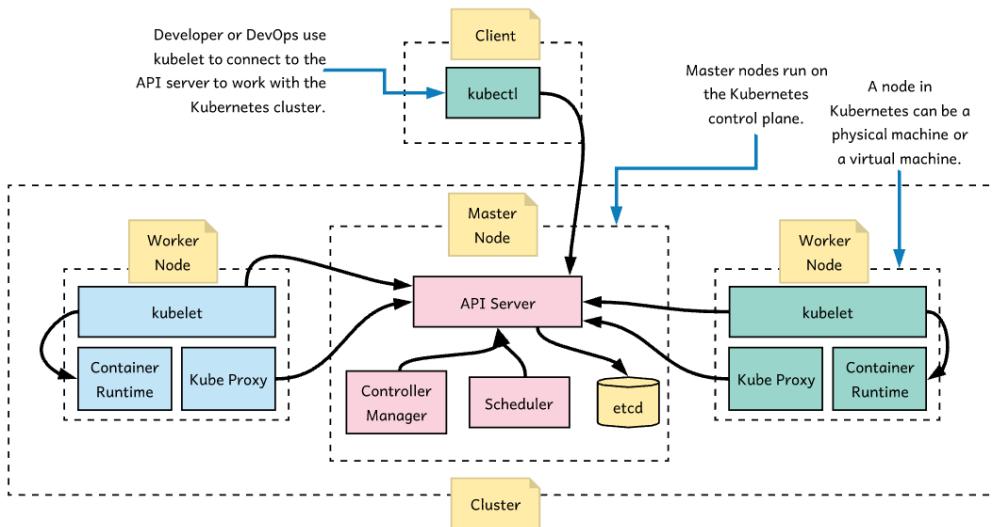


Figure B.1 A Kubernetes cluster consists of multiple master nodes and multiple worker nodes.

B.1.2 Worker node

Kubernetes runs workloads (containers) on worker nodes. When we instruct the master node to run a container (with a microservice) on Kubernetes, the master node then picks a worker node and instructs it to spin up and run the requested container. A worker node consists of three main components: kubelet, kube-proxy and the container runtime (see figure B.1). We discuss all these three components, in detail later in the appendix.

B.2 Basic constructs

To start working with Kubernetes, we need to understand some of its basic constructs. The following sections cover the most used constructs, but in no means a comprehensive a list.

B.2.1 A Pod, the smallest deployment unit in Kubernetes

A pod is the smallest deployment unit in a Kubernetes environment (see figure B.2). It is an abstraction over a group of containers – or in other words, a given Kubernetes pod can have more than one Docker container. In appendix A, we discussed Docker containers in detail, but in a Kubernetes environment we cannot deploy a container as it is – we need to wrap it within a pod. For example, if we are to deploy the Order Processing microservice in Kubernetes, we need to follow the steps below.

1. Create a Docker image for the Order Processing microservice.
2. Publish the Docker image to a Docker registry, which is accessible from the Kubernetes cluster.
3. Write a yaml file to describe the pod. This yaml file instructs Kubernetes, which Docker images it needs to pull from the Docker registry to create the pod. To be precise, we represent a pod, within a deployment. A Deployment is a Kubernetes object, which we discuss in section B.3.4; in section B.14, we discuss Kubernetes objects.
4. Use kubectl command-line tool to instruct the Kubernetes master node to create the pod.

Kubernetes uses yaml files to represent different constructs. According to the yaml specification (<https://yaml.org/spec/1.2/spec.html>), yaml is a human-friendly, cross language; Unicode based data serialization language designed around the common native data types of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing.

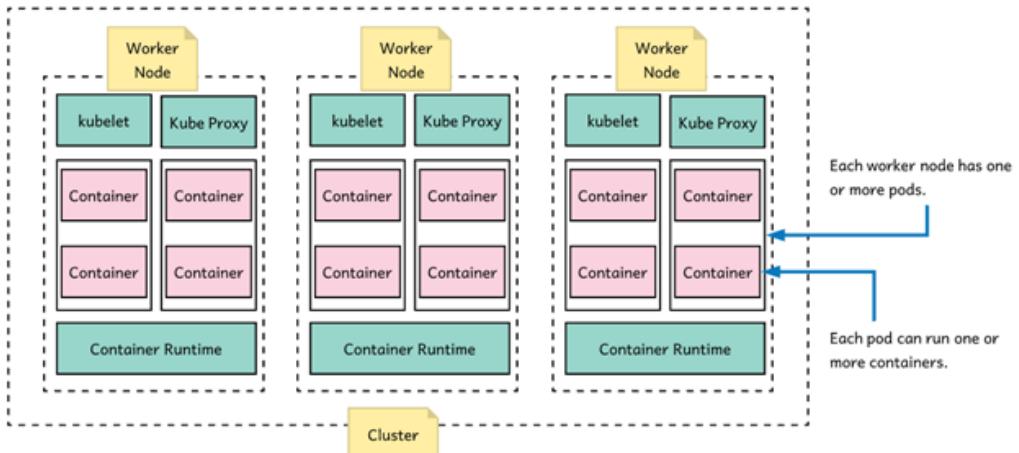


Figure B.2 A pod groups one or more containers and a worker node runs one or more pods.

Let's have a look at a sample yaml file, which describes a pod. There, to create the pod, Kubernetes has to pull the Docker image of the Order Processing microservice from Docker Hub (a public Docker registry), with the image name `prabath/manning-order-processing`.

Listing B.1. The definition of a Kubernetes Pod in yaml

```
apiVersion: v1
kind: Pod      #A
metadata:
  name: order-processing
  labels:
    app: order-processing
spec:
  containers:    #B
    - name: order-processing
      image: prabath/manning-order-processing      #C
      ports:
        - containerPort: 8080      #D
```

#A This describes a pod object

#B A given pod can have multiple containers

#C Name of the Docker container, the Kubernetes will pull from Docker Hub (registry)

#D The container listens on port 8080. If we run multiple containers on the same pod, they must have different ports.

B.2.2 A Node, a worker machine in Kubernetes

A node in Kubernetes is a virtual machine or a physical machine in a cluster. Or in other words, a Kubernetes cluster is a collection of Kubernetes nodes. When we instruct the Kubernetes master node to create a pod with one or more containers, it picks the most appropriate worker node and instructs it to run the pod. A given Kubernetes cluster has multiple nodes, but all the containers in a given pod, run in the same worker node. We can also instruct the Kubernetes master node to create multiple replicas of the same pod – and in that case these replicas will run on different nodes of the Kubernetes cluster. But still, for a given pod, all its containers run in the same node.

B.2.3 A Service, an abstraction over Kubernetes pods

Pods in Kubernetes are ephemeral or short-lived. They can come and go at any time. In other words, Kubernetes can start and stop a pod at any time. For example, when we create a pod, we can instruct Kubernetes to launch five instances (or replicas) of a pod to run all the time, but if the load (or the number of requests) coming to those pods goes beyond a certain threshold value, increase the number of pods to eight. This is how auto-scaling works in Kubernetes. Kubernetes will start creating more pods when the load crosses up a given threshold and stop certain pods when the load crosses down the threshold. Also, we can ask Kubernetes to run a minimal set of pods all the time (no matter what), then if a pod goes down by itself (crashes), Kubernetes will still make sure it spins up a new one to maintain the minimal number of pods we asked it to run.

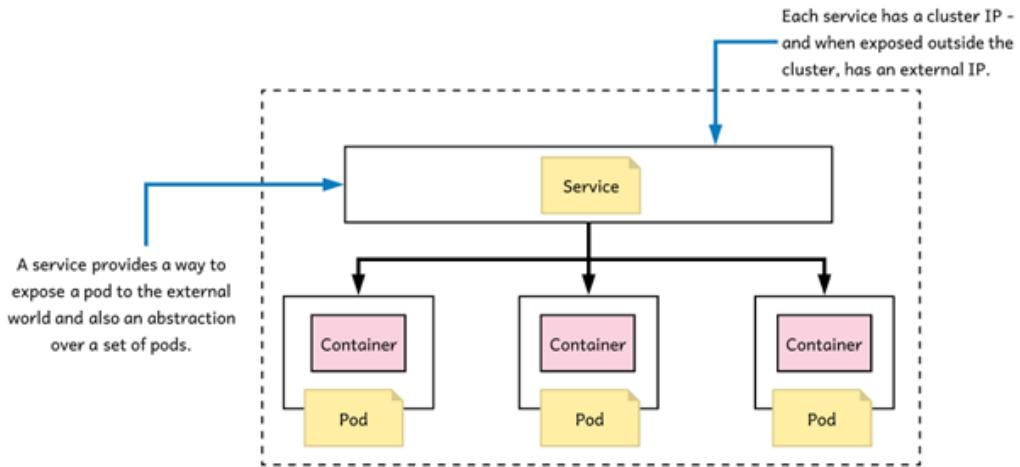


Figure B.3 A service groups one or more pods and exposes pods outside of the Kubernetes cluster. Not all the services are accessible from outside a Kubernetes cluster.

Since pods are ephemeral, the IP address assigned to a pod can change over time and at the same time we cannot exactly predict in an auto-scaling environment, how many pods would be running at a given time and what are those. Due to these reasons we cannot directly communicate with pods, instead use a service. A service is an abstraction over a set of pods (see figure B.3). You can create a Kubernetes service pointing to a set of pods. You can think of a service as a way to route requests to a pod.

Then again, if the pod IP address is changing over the time and if a pod can come and go at any time, how do we bind a service to a pod or to a set of pods? When we create a service against a set of pods, we don't create a static binding between them or in other words, we don't tell the service that these are the IP addresses of the pods you need to work with. Instead, we use a label to define a filtering criterion, so that Kubernetes can filter out the pods that have to work with a given service. If you look at the sample yaml shown in listing B.2, you will find a label assigned to that pod, which is `app:order-processing`. A label is a key value pair – `app` is the key and `order-processing` is the value. All the replicas created from that yaml file or all the replicas corresponding to that pod will carry the same label. If we want to create a service pointing to all the replicas of that pod, then we can use that label as a filter criterion, when defining the service.

Listing B.2. The definition of a Kubernetes Pod in yaml

```
apiVersion: v1
kind: Pod
metadata
  name: order-processing
  labels:
    app: order-processing
```

```

spec:
  containers:
    - name: order-processing
      image: prabath/manning-order-processing
      ports:
        - containerPort: 8080

```

CLUSTERIP SERVICE

When you create a service in Kubernetes without specifying any type (or setting the type as ClusterIP), Kubernetes creates a service of cluster IP type or in other words, cluster IP is the default service type in Kubernetes. A service of cluster IP type is only reachable within a Kubernetes cluster. The listing B.3 shows a sample yaml of a Kubernetes service of ClusterIP type.

Listing B.3. The definition of Cluster IP Service in yaml

```

apiVersion: v1
kind: Service
metadata:
  name: order-processing-service
spec:
  type: ClusterIP    #A
  selector:    #B
    app: order-processing
  ports:
    - protocol: TCP
      port: 80    #C
      targetPort: 8080    #D

```

#A Type is an optional attribute defines the type of the service. If no type specified then it is a ClusterIP service.

#B The selector selects the set of pods in the Kubernetes cluster by the matching labels.

#C The service port, which is accessible along with the cluster IP.

#D Each pod corresponding to this service, listens on the targetPort

NODEPORT SERVICE

A Kubernetes cluster has multiple nodes. When we want to expose a microservice over multiple pods as a NodePort Service, Kubernetes opens up a port on each node – and it is exactly the same port. To access a NodePort service from a client outside the cluster, we need to use a node IP address (each node in the cluster has its own IP) and the port. When a node gets a request on a particular port (which is called the `nodePort`), which is bound to a NodePort Service, it routes the request to any of its pods. This type of a Service does not have a single external IP address, but only a cluster IP address, which is only accessible within the cluster. If you want to access a NodePort service within the Kubernetes cluster, you need to use the cluster IP address and the corresponding service port (`port` – not the `nodePort`). For clients out side the Kubernetes cluster, a NodePort service is accessible via any IP address belongs to any node in the cluster on the node port (`nodePort`). If a given node is down, then the client has to detect the failure and switch to a different node. That is one disadvantage in this approach. Following shows a sample yaml of a Kubernetes Service of NodePort type. The listing B.4 shows a sample yaml of a Kubernetes service of NodePort type.

Listing B.4. The definition of NodePort Service in yaml

```

apiVersion: v1
kind: Service
metadata:
  name: order-processing-service
spec:
  type: NodePort    #A
  selector:    #B
    app: order-processing
  ports:
  - protocol: TCP
    port: 80      #C
    targetPort: 8080    #D
    nodePort: 30200   #E

```

#A Type is an optional attribute defines the type of the service. If no type specified then it is a ClusterIP service.

#B The selector selects the set of pods in the Kubernetes cluster by the matching labels.

#C The service port, which is accessible along with the cluster IP.

#D Each pod corresponding to this service, listens on targetPort

#E Each node in the cluster listens on the same nodePort

The above yaml creates a service pointing to all the pods carrying the label, `app:order-processing`. Each service has an internal IP address (which is called a cluster IP), which is only accessible within the cluster. As per listing B.4, this Service listens on port 80 on the internal IP address. Any pod to access the Service, within the cluster, can use the service's internal IP address and the port (80). The nodePort (30200) is the port every node in the cluster listens to any traffic coming on node IP address (not the cluster IP) and routes the to the port 8080 (targetPort) of the corresponding pod. If we do not specify a nodePort, while creating the service, Kubernetes will internally pick an appropriate port.

LOADBALANCER SERVICE

The LoadBalancer service type is an extension of the NodePort service type. If there are multiple replicas of a given pod, the service of LoadBalancer type can act as a load balancer. Usually it is an external load balancer provided by the Kubernetes hosting environment. In case Kubernetes hosting environment does not support services of LoadBalancer type, a service defined as a LoadBalancer will still run fine, but as a service of NodePort. Following shows a sample yaml of a Kubernetes service of LoadBalancer type. The listing B.5 shows a sample yaml of a Kubernetes service of LoadBalancer type.

Listing B.5. The definition of LoadBalancer Service in yaml

```

apiVersion: v1
kind: Service
metadata:
  name: order-processing-service
spec:
  type: LoadBalancer    #A
  selector:    #B
    app: order-processing
  ports:

```

```

- protocol: TCP
  port: 80
  targetPort: 8080

```

#A Type, optional attribute defines the type of the service. If no type specified then it is a ClusterIP service.

#B The selector selects the set of pods in the Kubernetes cluster by the matching labels.

The yaml in listing B.5 creates a service pointing to all the pods carrying the label, app:order-processing. The service listens on port 80 (load balancer port) and reroutes traffic to port 8080 of the corresponding pod. Here the rerouting works via a nodePort (which we discussed in section B.2.2). Even though we do not define a nodePort in listing B.5, Kubernetes automatically generates one. So the way routing works precisely is, the external load balancer listens on port 80 and routes the traffic it gets to the nodePort of any of a node that carries the corresponding pod – and then the nodePort reroutes the traffic to port 8080 of the pod. Multiple services can listen on the same port, but each service has its own public IP address, pointing to the load balancer.

B.2.4 A Deployment, represents your application in Kubernetes

We discussed about pods under the section B.2.1. Even though the pod concept is fundamental to Kubernetes, in practice we do not deal with pods directly (but Kubernetes does). We as developers (or DevOps) deal with Kubernetes deployments. A deployment represents your application, which carries multiple replicas of a given pod. A deployment is a Kubernetes object that helps managing pods. A given deployment can only manage one pod definition (there can be multiple replicas, but still one pod definition). We can use a deployment to create and scale a pod. It also helps you migrate your application from one version to another, following a migration strategy you pick (blue/green, canary and so on). Instead of the yaml file we used under section B.2.1 (listing B.1) to create a pod, we can use the following yaml in listing B.6 to create a deployment.

Listing B.6. The definition of Kubernetes Deployment in yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-processing-deployment
  labels:
    app: order-processing
spec:
  replicas: 5      #A
  selector:
    matchLabels:
      app: order-processing      #B
  template:        #C
    metadata:
      labels:
        app: order-processing
    spec:
      containers:
        - name: order-processing

```

```
image: prabath/manning-order-processing
ports:
- containerPort: 8080
```

#A Instructs Kubernetes to run 5 replicas of the matching pods.
#B This deployment will carry a matching pod as per the selector. This is an optional section, which can carry multiple labels.
#C A template describes how each pod in the deployment should look like. If the deployment has defined a selector/matchLabels, then the pod definition must carry a matching label.

B.2.5 A namespace, your home within a Kubernetes cluster

A Kubernetes namespace is a virtual cluster within the same physical Kubernetes cluster. One Kubernetes cluster can have multiple namespaces. In practice organizations have one Kubernetes cluster with different namespaces for different environments. For example, one namespace is used for the development, one for testing, one for staging, and another one for production. Also some organizations use two different Kubernetes clusters, one for production environments and another one for pre-production. The production Kubernetes cluster has one namespace for staging and another for production. The pre-production cluster has namespaces for development and testing. Following yaml file represents a Kubernetes namespace.

```
apiVersion: v1
kind: Namespace
metadata:
  name: manning
```

A Kubernetes namespace has following characteristics:

- The Kubernetes object names (for example the name of a pod, a service, a deployment), must be unique within a namespace, but not across namespaces. In section B.13 we discuss Kubernetes objects in detail.
- The names of namespaces, nodes and persistent volumes must be unique across all the namespaces in a cluster. A persistent volume (PersistentVolume) in Kubernetes provides an abstraction over storage.
- By default, a pod in one namespace can talk to another pod in a different namespace. To prevent such, we need to use some Kubernetes plugins to bring in network isolation by namespaces.
- Each namespace can have its own resource allocation. For example, when you share the same Kubernetes cluster for development and production, with two different namespaces, we can allocate more CPU cores and memory to the production namespace.
- Each namespace can have a limit based on the number of objects. For example, the development namespace can go up to 10 pods and 2 services, while the production namespace can go up to 50 pods and 10 services.

B.3 Getting started with Minikube

Minikube provides a single-node Kubernetes cluster that can run on your local machine. It has certain limitations related to scalability, but is one of the easiest ways to get started with

Kubernetes. The online Kubernetes documentation available at <https://kubernetes.io/docs/setup/minikube/> provides all necessary steps in setting up minikube.

B.4 Kubernetes as a service

Instead of running Kubernetes on your own servers and maintaining the hardware yourself, you can look for a cloud vendor who is providing Kubernetes as a service. There are multiple cloud vendors doing that and we see most of the new Kubernetes deployments rely on cloud vendors.

- Google provides Kubernetes as a service with Google Kubernetes Engine (GKE).
- Amazon runs a Kubernetes as a service on AWS, which is known as Amazon Elastic Container Service for Kubernetes (EKS).
- Azure Kubernetes Service (AKS) is the Kubernetes as a service implementation by Microsoft.
- OpenShift Container Platform (OCP) is the Kubernetes as a service managed by Red Hat.
- Pivotal Container Service (PKS) is the Kubernetes as a service managed by Pivotal.
- IBM Cloud Kubernetes Service is the Kubernetes as a service managed by IBM.
- Container Engine for Kubernetes is the Kubernetes as a service managed by Oracle.
- VMware Cloud PKS is the Kubernetes as a service managed by VMware.

Each of the above cloud platforms has it's own pros and cons, but still the fundamental concepts around Kubernetes remain unchanged. In this book we use Google Kubernetes Engine (GKE) for all the samples.

B.5 Getting started with Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) is the Kubernetes as a service implementation managed by Google. To get started, you need to have a valid Google account and it gives you \$300 credit for free to try out Google Cloud Platform. It's more than enough to try out all the samples in the book. You can sign up for the free trial with the link (or Google for 'GKE free trial'): <https://cloud.google.com/free/>. Then follow the straightforward instructions available at <https://cloud.google.com/kubernetes-engine/docs/quickstart> to get started with GKE. These instructions may subjected to change over time, so we avoid repeating them in the book – always refer to the GKE online documentation to get started.

B.5.1 Installing gcloud

Google Cloud Platform (GCP) provides a command-line tool to interact with your GKE running on the cloud, from your local machine. Follow the instructions available at <https://cloud.google.com/sdk/docs/quickstarts>, corresponding to your operating system to install gcloud. In fact what you install is the Google Cloud SDK, and gcloud is part of it. Once you have successfully installed gcloud, run the following command to make sure everything is working fine.

```
\> gcloud info
```

```

Google Cloud SDK [247.0.0]
Python Version: [2.7.10 (default, Feb 22 2019, 21:17:52) [GCC 4.2.1 Compatible Apple LLVM
10.0.1 (clang-1001.0.37.14)]]
Installed Components:
  core: [2019.05.17]
  gsutil: [4.38]
  bq: [2.0.43]
Account: [prabath@wso2.com]
Project: [kubetest-232501]
Current Properties:
  [core]
    project: [kubetest-232501]
    account: [prabath@wso2.com]
    disable_usage_reporting: [False]
  [compute]
    zone: [us-west1-a]

```

The output above only shows some important parameters. When you run the same command in your local setup, you will get a different result with more parameters.

B.5.2 Installing kubectl

To interact with the Kubernetes environment running on Google Cloud, we also need to install kubectl, as a component of the gcloud tool, which we installed in B.5.1. kubectl is a command line tool, which runs on your local computer and talks to Kubernetes API server running on Google Cloud to perform certain operations. Following command installs kubectl as a component of the gcloud tool.

```
\> gcloud components install kubectl
```

To verify the kubectl installation, run the following command, which should result in some meaningful output, with no errors.

```
\> kubectl help
```

B.5.3 Setting up default setting for gcloud

The gcloud command line tool has an option that it can remember certain settings, so each time when you run a gcloud command; you do not need to repeat them. Following example sets the default GKE project id. It is an identifier associated with a GKE project that you create from the web-based console. At the time you set up your GKE account, you also created a project. You need to replace [PROJECT_ID] with your own project id.

```
\> gcloud config set project [PROJECT_ID]
```

Following command sets the default region or the compute zone. For example, when you create a Kubernetes cluster in GKE, you need to specify under which region you need to create it. All the resources associated with that Kubernetes cluster would live in that particular region. In other words, it's a geographical location. Following command sets the default compute zone to us-west1-a (The Dalles, Oregon, USA). In fact, us-west1 is the region and a is the zone. A zone is an isolated location within a region, which defines the capacity and the type of the

available resources. For example, us-west1 region has three zones: a, b, c and zone a, has Intel Xeon E5 v4 (Broadwell) platform by default, with up to 96 vCPU machine types on Skylake platform. More details on GKE regions and zones are documented here: <https://cloud.google.com/compute/docs/regions-zones/>.

```
\> gcloud config set compute/zone us-west1-a
```

B.5.4 Creating a Kubernetes cluster

Before we do anything on Kubernetes, first we need to create a cluster. This is not something we do quite frequently. Following command creates a Kubernetes cluster using the gcloud command-line tool with the name manning-ms-security.

```
\> gcloud container clusters create manning-ms-security

Creating cluster manning-ms-security in us-west1-a... Cluster is being configur
ed...
Creating cluster manning-ms-security in us-west1-a... Cluster is being deployed
...
Creating cluster manning-ms-security in us-west1-a... Cluster is being health-checked (master
is healthy)...done.
Created [https://container.googleapis.com/v1/projects/kubetest-232501/zones/us-west1-
a/clusters/manning-ms-security]
```

Once the Kubernetes cluster is created successfully, we need to configure the kubectl command-line tool to work with the cluster. Following command fetches the authentication credentials to connect to the cluster (manning-ms-security) and configures kubectl.

```
\> gcloud container clusters get-credentials manning-ms-security
```

Now let's use the following kubectl command to find the version of Kubernetes client and server. All the commands we run with kubectl tool are not specific to GKE, but common across all Kubernetes deployments.

```
\> kubectl version
```

B.5.5 Deleting a Kubernetes cluster

You can use the following gcloud command to delete a Kubernetes cluster (manning-ms-security) created on GKE, but lets not do it till we finish the samples in the book.

```
\> gcloud container clusters delete manning-ms-security
```

B.6 Creating a Kubernetes deployment

There are two ways to create a Kubernetes deployment. One way is by using a yaml file and the other way is simply using the kubectl command-line tool. Even if we use the yaml file, we will still use the kubectl to communicate with the Kubernetes cluster running in the cloud. In a production deployment, we use yaml files to maintain Kubernetes deployment configuration. In most of the cases, these yaml files are versioned and maintained in a git repository. A detailed discussion on a Kubernetes deployment is out of scope of this book, and for any readers

interested in learning more, we would recommend reading the book *Kubernetes in Action* (Manning Publications, 2017) by Marko Lukša. Also, the book *Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications* (O'Reilly Media, 2019) by Bilgin Ibryam and Roland Huß is a very good reference to learn how to use Kubernetes in a production deployment. In chapter 11, in all the samples we use yaml files to create Kubernetes deployments, but in this appendix we use the command line options.

Let's use the following kubectl command to create a Kubernetes deployment, with the Docker image: `gcr.io/google-samples/hello-app`. If you run the same command again and again, you will get an error saying: `Error from server (AlreadyExists): deployments.apps "hello-server" already exists`. In that case you need to delete the deployment before running the kubectl run command again (check the end of this section for the delete command).

```
\> kubectl run hello-server --image gcr.io/google-samples/hello-app:1.0 --port 8080
deployment.apps/hello-server created
```

When you run the above command, Kubernetes will fetch the Docker image from gcr.io Docker registry and run it as a container on the Kubernetes cluster we just created. The `port` argument in the kubectl command specifies that the process running in the container be exposed over port 8080. Now, if you run the following command, it will show you all the deployments in the current Kubernetes cluster (under the default namespace).

```
\> kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-server   1         1         1           1          13s
```

If you would like to see the yaml representation of the above Kubernetes deployment, we can use the following command, which carries the value `yaml` for the `-o` (output) argument. This will result in a lengthy output, which carries all the details related to the `hello-server` deployment.

```
\> kubectl get deployments hello-server -o yaml
```

Under section B.2.1 we discussed that, a pod is the smallest deployment unit in a Kubernetes environment. When we create a deployment, the related pods get created automatically and still the containers are running inside a pod. We can use the following kubectl command to list out all the pods running in our Kubernetes cluster.

```
\> kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
hello-server-5cdf4854df-q42c4   1/1     Running   0          10m
```

If you'd like to delete the deployment we created above, we can use the following command. But let's not do it, till we finish this appendix.

```
\> kubectl delete deployments hello-server
```

B.7 Behind the scenes of a deployment

When we create a deployment, it internally creates another object called, ReplicaSet. We differed this discussion, when we introduced the Deployment object in section B.2.4 for simplicity. As developers or DevOps, we do not directly deal with the ReplicatSet, but Kubernetes internally does. In a Deployment, the pods are created and managed by a ReplicaSet. The following kubectl command lists out all the ReplicaSets in the Kubernetes cluster (under the default namespace).

```
: \> kubectl get replicasesets
NAME          DESIRED  CURRENT  READY  AGE
hello-server-5cdf4854df  1        1        1      11m
```

The following kubectl command in listing B.7 gets more details corresponding to the hello-server-5cdf4854df ReplicaSet and prints the output in yaml format.

Listing B.7. The definition of a ReplicaSet in yaml

```
: \> kubectl get replicasesets hello-server-5cdf4854df -o yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  labels:
    run: hello-server
  name: hello-server-5cdf4854df
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    kind: Deployment
    name: hello-server
    uid: c7460660-7d38-11e9-9a8e-42010a8a014b
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-server
  template: #A
    metadata:
      labels:
        run: hello-server
    spec:
      containers:
      - image: gcr.io/google-samples/hello-app:1.0
        name: hello-server
      ports:
      - containerPort: 8080
        protocol: TCP
status:
  availableReplicas: 1
```

#A Defines the pod, which this ReplicaSet controls.

The truncated output above only shows some important sections and attributes. The spec/template section defines the pod, which this ReplicaSet manages.

B.8 Creating a Kubernetes Service

The `hello-server` deployment (which we created under the section B.6), listens on port 8080. It is not accessible outside the Kubernetes cluster. In other words, it does not have an IP address, which is accessible outside the cluster. In Kubernetes, a container that carries a microservice (or in our case `hello-server`) is deployed in a pod and the communication happens between pods. One pod can talk to another pod. Each pod in a Kubernetes environment has an IP address. You can run the following `kubectl` commands to get more information about a pod, running within a deployment.

```
\> kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
hello-server-5cdf4854df-q42c4  1/1     Running   0          10m

\> kubectl describe pod hello-server-5cdf4854df-q42c4
NAME                  READY   STATUS    RESTARTS   AGE
hello-server-5cdf4854df-q42c4  1/1     Running   0          10m

Name:            hello-server-5cdf4854df-q42c4
Namespace:       default
Status:          Running
IP:              10.36.0.6
```

The above command shows all the details related to the provided pod (`hello-server-5cdf4854df-q42c4`), but for clarity only few important parameters are shown in the output – most importantly the IP address. This IP address assigned to a pod by Kubernetes is only accessible within the same cluster. In a typical Kubernetes deployment for a given microservice (or a container), we run multiple instances of the same pod to address scalability requirements. In other words, in a given Kubernetes cluster, a given pod has multiple replicas. This will help Kubernetes to distribute the requests coming to a given microservice, between all the corresponding pods – or do load balancing. Pods in Kubernetes are ephemeral or short-lived. They can come and go at anytime – so even the internal IP address assigned to a pod can change time to time. This is one requirement for having a Kubernetes Service (see section B.2.3). A Kubernetes Service provides an abstraction over a set of pods, that matches with a given criteria. You don't talk to a pod directly, but always go through a service. Let's use the following command to create a Kubernetes Service.

```
\> kubectl expose deployment hello-server --type LoadBalancer --port 80 --target-port 8080
```

The above command creates a Kubernetes Service over all the pods running in the `hello-server` deployment. Here we create a service of the type LoadBalancer. Let's use the following command to discover all the services running in our Kubernetes cluster (under the default namespace).

```
\> kubectl get services
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hello-server  LoadBalancer  10.39.243.41  35.233.245.242  80:30648/TCP  6h8m
kubernetes  ClusterIP  10.39.240.1    <none>        443/TCP      8h
```

The above output shows that `hello-server` Service has two IP addresses. When we access a Service within a Kubernetes cluster, we use the cluster IP. And for accessing a service from an external client we use the external IP. Kubernetes may take few minutes to assign an external IP address to a Service. If you don't see an IP address assigned to your Service, repeat the command in few minutes. We can use the following curl command to test the `hello-server` running in GKE, with the external IP.

```
\> curl http://35.233.245.242
Hello, world!
Version: 1.0.0
Hostname: hello-server-5cdf4854df-q42c4
```

B.9 Behind the scenes of a Service

When we create a Service, behind the scenes, Kubernetes creates an Endpoints object corresponding to the Service. Before we delve deep into why we need Endpoints, let's use the following kubectl command to list out all the Endpoints objects in the Kubernetes cluster (under the default namespace).

```
\> kubectl get endpoints
NAME           ENDPOINTS      AGE
hello-server   10.36.0.6:8080   5d18h
```

Each Endpoints object carries a set of pod IP addresses and the corresponding container ports, with respect to the pods associated with a given Service. Following kubectl command will find more details of the `hello-server` Endpoints object. The truncated output in listing B.8 shows the definition of an Endpoints object, with respect to the Service, associated with 3 replicas of a given pod.

Listing B.8. The truncated definition of an Endpoints object

```
apiVersion: v1
kind: Endpoints
metadata:
  name: hello-server
  namespace: default
subsets:
- addresses:
  - ip: 10.36.0.6    #A
    nodeName: gke-manning-ms-security-default-pool-6faf40f5-5cnb    #B
    targetRef:
      kind: Pod
      name: hello-server-5cdf4854df-q42c4    #C
- ip: 10.36.1.36
  nodeName: gke-manning-ms-security-default-pool-6faf40f5-br1m
  targetRef:
    kind: Pod
    name: hello-server-5cdf4854df-5z8hj
- ip: 10.36.2.28
  nodeName: gke-manning-ms-security-default-pool-6faf40f5-vh0x
  targetRef:
    kind: Pod
    name: hello-server-5cdf4854df-bdgdc
```

```
ports:
  - port: 8080
    protocol: TCP
```

#A IP address pointing to a pod

#B The name of the node, which runs the pod. Here we can see each pod runs on a different node.

#C The name of the pod.

As we discussed in B.9, a Service finds the associated pods, by matching the corresponding labels. At the runtime, for a service to route traffic to pod, it has to know the pod IP address and the port. To address this need, once we find all the pods associated with a given service, Kubernetes creates an Endpoints object to carry all the pod IP addresses and the corresponding container ports associated with that Service. The Endpoints object gets updated all the time whenever there is a change in a corresponding pod.

B.10 Scaling a Kubernetes deployment

In our `hello-server` deployment so far, we only have one pod. We can use the following `kubectl` command to ask Kubernetes to create 5 replicas of the pod, within the same deployment. Kubernetes will make sure it always maintains 5 replicas of the pod – and in case one pod goes down, it will spin up a new one.

```
\> kubectl scale --replicas 5 deployment hello-server
deployment.extensions/hello-server scaled

\> kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
hello-server-5cdf4854df-c9b4j  1/1     Running   0          52s
hello-server-5cdf4854df-fs6hg  1/1     Running   0          53s
hello-server-5cdf4854df-hpc7h  1/1     Running   0          52s
hello-server-5cdf4854df-q42c4  1/1     Running   0          7h24m
hello-server-5cdf4854df-qjkgp 1/1     Running   0          52s
```

B.11 Creating a Kubernetes namespace

As we learnt in section B.3.6, a Kubernetes namespace is a virtual cluster within the same physical Kubernetes cluster. Kubernetes comes with its own set of namespaces. Let's use the following `kubectl` command to view available Kubernetes namespaces.

```
\> kubectl get namespaces
NAME      STATUS   AGE
default   Active   23h
kube-public   Active   23h
kube-system   Active   23h
```

Each object we create in Kubernetes belongs to a namespace. All the objects created by the Kubernetes system itself belong to the `kube-system` namespace. An object belonging to one namespace is not accessible from another namespace. If we want to have some objects accessible from any namespace, then we need to create them in `kube-public` namespace. When

we create an object with no namespace, then those objects belong to the default namespace. Let's use the following kubectl command to create a custom namespace called manning.

```
: \> kubectl create namespace manning
namespace/manning created
```

The following kubectl command shows how to create a deployment in the manning namespace. If we skip the --namespace argument in the command, then Kubernetes will assume it is the default namespace.

```
: \> kubectl run manning-hello-server --image gcr.io/google-samples/hello-app:1.0 --port 8080
--namespace=manning
```

The following kubectl command shows all the deployments in the default namespace (as we do not mention a namespace, Kubernetes assumes it is the default one) – and does not show the manning-hello-server deployment we created from the previous command.

```
: \> kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-server   5         5         5           5          22h
```

To view all the deployments under the manning namespace lets use the following kubectl command (instead of --namespace, you can also use -n).

```
: \> kubectl get deployments --namespace manning
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
manning-hello-server   1         1         1           1          36s
```

B.12 Switching Kubernetes namespaces

If we are to work on multiple Kubernetes namespaces, sometimes it's a pain to switch between them. The kubectl tool let's you switch between multiple Kubernetes clusters and namespaces quite easily. Installation of kubectx is quite straightforward, when you follow the instructions listed here: <https://github.com/ahmetb/kubectx>. Once the installation is done, we can use the following kubens command (which comes with kubectx) to set manning as the default namespace, so we do not need to pass the namespace argument with each and every kubectl command.

```
\> kubens manning
Context "gke_kubetest-232501_us-west1-a_manning-ms-security" modified.
Active namespace is "manning".
```

B.13 Kubernetes objects

Kubernetes has a rich object model to represent different aspects of an application running in a distributed environment. Using a yaml file we can describe each of these Kubernetes objects and create, update and delete using an API, which is exposed by the Kubernetes API server –

and persist in etcd². For example, a pod, a service, a volume and a namespace are basic Kubernetes objects, which we discussed in section B.2. Each object has an attribute called apiVersion and kind – and another set of attributes under three main categories: metadata, spec and status.

- API version (apiVersion) defines the versioned schema of the object representation. This helps to avoid any conflicts in version upgrades.
- Kind is a string, which represents the type of the object. For example the value of kind attribute can be Pod, Deployment, Namespace, Volume and so on.
- Metadata includes the name of the object, a unique identifier (UID), the namespace of the object and other standard attributes.
- The spec describes the desired state of a Kubernetes object at the time you create (or update) it.
- The status represents the actual state of the object at the runtime.

The difference between spec and status may not be quite clear – so let's go through an example. Let's revisit the command we used to create a deployment in section B.6. It uses the following kubectl command to create a Kubernetes Deployment, with the Docker image: gcr.io/google-samples/hello-app. This is an example of an imperative command. With an imperative command, we ask Kubernetes exactly what to do and give all the required parameters to manage a given Kubernetes object as command-line arguments (under section B.13.1 we learn about creating Kubernetes objects using declarative configuration as opposed to imperative commands).

```
\> kubectl run hello-server --image gcr.io/google-samples/hello-app:1.0 --port 8080
deployment.apps/hello-server created
```

The above command creates a Deployment object in the Kubernetes cluster. We can use the following kubectl command to get the definition of it and only a truncated output is shown below.

```
\> kubectl get deployment hello-server -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-server
spec:
  replicas: 1
status:
  availableReplicas: 1
```

Here we can see the number of replicas under spec category is 1, which is the required number of replicas and the availableReplicas under status category is also 1. Let's use another kubectl imperative command to increase the number of replicas to 100. Once again this

²The etcd is a highly available key-value store, which Kubernetes uses to persist cluster data. API server persists the data related to Kubernetes objects in etcd.

is exact what happens when we use imperative commands, we need to tell Kubernetes how to do everything – not just what we need.

```
: \> kubectl scale --replicas 100 deployment hello-server
deployment.extensions/hello-server scaled
```

The above command creates 100 replicas of the deployment, which would probably take few seconds. We can use the following kubectl command few times just after a second from the previous command to get the definition of the deployment and only a truncated output is shown in listing B.9.

Listing B.9. The truncated definition of hello-server Deployment object

```
\> kubectl get deployment hello-server -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-server
spec:
  replicas: 100
status:
  availableReplicas: 1

\> kubectl get deployment hello-server -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-server
spec:
  replicas: 100
status:
  availableReplicas: 14
```

Here we can see the number of replicas under spec category is 100 all the time, while the availableReplicas under status category is 1 initially and then when we run the same command for the second time, availableReplicas under status category has increased to 14. Attributes under spec category represents the requirements, while the attributes under status category represent the actual runtime status of a Kubernetes object.

B.13.1 Managing Kubernetes objects

We can use kubectl to manage Kubernetes objects, in three ways: imperative commands, imperative object configuration, and declarative object configuration. In the previous section we used imperative commands. It is a great way to get started, but in a production deployment we should not use. Imperative commands change the state of the Kubernetes objects in a cluster with no tracing. For example, if we want to revert a change we did, we need to do it manually by remembering the command we executed.

IMPERATIVE OBJECT CONFIGURATION

With imperative object configuration, we use a yaml file to represent a Kubernetes object (instead of passing all required attribute as command-line arguments) and then use kubectl to

manage it. For example, let's have a look at the following content in the file, `hello-server.yaml` (listing B.10). It represents the `hello-server` deployment.

Listing B.10. The definition of hello-server Deployment object

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-server
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-server
  template:
    metadata:
      labels:
        run: hello-server
    spec:
      containers:
        - image: gcr.io/google-samples/hello-app:1.0
          name: hello-server
          ports:
            - containerPort: 8080
```

Let's use the following `kubectl` command to create a deployment using the above object configuration. If you have run the previous examples in this appendix, you already have the `hello-server` deployment. In that case, change `metadata/name` in the above configuration to some other name. You can find the `hello-server.yaml` file inside the `appendix-b/sample01` directory of the samples repo of the book available at <https://github.com/microservices-security-in-action/samples/>

```
\> kubectl create -f hello-server.yaml
deployment.extensions/hello-server created
```

To update a Kubernetes object use the following `kubectl` command with the updated object configuration.

```
\> kubectl replace -f hello-server.yaml
```

One drawback here is, the updated `hello-server.yaml` must carry all the attributes that are necessary to replace the object. You cannot do a partial update.

DECLARATIVE OBJECT CONFIGURATION

With declarative object configuration we don't need to tell Kubernetes how to do things, but only what we need. To create a deployment, first we need to have a yaml file to represent it (just like in imperative object configuration), let's use the same `hello-server.yaml`, as in listing B.10. Let's use the following `kubectl` command to create a deployment using the object configuration in listing B.10. If you have run the previous examples in this appendix, you already have the `hello-server` deployment. In that case, change `metadata/name` in the above configuration to some other name.

```
: \> kubectl apply -f hello-server.yaml
```

When we run the above command, we do not ask Kubernetes to create or update the deployment – we just specify the characteristics of the Deployment we need and Kubernetes automatically detects the required operations and executes those. Whether we want to create a new object or update an existing one, still we use the same `apply` command.

B.14 Exploring Kubernetes API server

Kubernetes API server runs on Kubernetes control plane. Let's see how we can directly connect to the API server and discover what APIs are hosted there.

```
\> kubectl proxy &
Starting to serve on 127.0.0.1:8001
```

The above `kubectl` command will open up a connection to the API server. It in fact spins up a proxy server in your local machine, by default listening on port 8001. Then we can run the following `cURL` command in listing B.11 to figure out all the API paths hosted on the API server – and listing B.11 only shows a truncated output.

Listing B.11. The list of APIs hosted in the Kubernetes API server

```
: \> curl http://localhost:8001/
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/apis/apps",
    "/apis/apps/v1",
    "/apis/authentication.k8s.io",
    "/apis/authorization.k8s.io/v1",
    "/apis/authorization.k8s.io/v1beta1",
    "/apis/autoscaling",
    "/apis/batch",
    "/apis/batch/v1",
    "/apis/batch/v1beta1",
    "/apis/certificates.k8s.io",
    "/apis/certificates.k8s.io/v1beta1",
    "/apis/cloud.google.com",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/apis/metrics.k8s.io",
    "/apis/metrics.k8s.io/v1beta1",
    "/apis/networking.gke.io",
    "/apis/policy",
    "/apis/policy/v1beta1",
    "/apis/rbac.authorization.k8s.io",
    "/apis/rbac.authorization.k8s.io/v1",
    "/apis/scheduling.k8s.io",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
```

```

        "/swagger-2.0.0.json",
        "/swagger-2.0.0.pb-v1",
        "/swagger-2.0.0.pb-v1.gz",
        "/swagger.json",
        "/swaggerapi",
        "/version"
    ]
}

```

We can use the following cURL command in listing B.12 to list all the resources supported by a given Kubernetes API version – and listing B.12 only shows a truncated output.

Listing B.12. The list of resources available under the Kubernetes API version 1

```
\> curl http://localhost:8001/api/v1
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
      "name": "configmaps",
      "singularName": "",
      "namespaced": true,
      "kind": "ConfigMap",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch"
      ],
      "shortNames": [
        "cm"
      ]
    },
    {
      "name": "endpoints",
      "singularName": "",
      "namespaced": true,
      "kind": "Endpoints",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch"
      ],
      "shortNames": [
        "ep"
      ]
    }
  ]
}
```

```
{
  "name": "pods",
  "singularName": "",
  "namespaced": true,
  "kind": "Pod",
  "verbs": [
    "create",
    "delete",
    "deletecollection",
    "get",
    "list",
    "patch",
    "update",
    "watch"
  ],
  "shortNames": [
    "po"
  ],
  "categories": [
    "all"
  ]
}
]
```

B.15 Kubernetes resources

A resource is an instance of a Kubernetes object, which is accessible by a unique URL or in other words it's a REST resource. We can access the same Kubernetes object via multiple resource URLs. For example, (assuming that you have the `hello-server` deployment still running), the following `kubectl` command retrieves the definition of the `hello-server` deployment as a yaml output.

```
\> kubectl get deployment hello-server -o yaml
```

Following shows the truncated output of the above command, which only includes some important attributes.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-server
  namespace: default
  resourceVersion: "1137529"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/hello-server
  uid: c7460660-7d38-11e9-9a8e-42010a8a014b
spec:
  replicas: 1
status:
  availableReplicas: 1
```

The `selfLink` attribute in the above output is the resource that represents the `hello-server` deployment. Like we discussed in section B.14 – we can start a proxy locally and do `cURL` to this resource URL and get the complete representation of the `hello-server` Deployment resource.

```
\> kubectl proxy &
Starting to serve on 127.0.0.1:8001

\> curl http://localhost:8001/apis/extensions/v1beta1/namespaces/default/deployments/hello-
server
```

B.16 Kubernetes controllers

In section B.13 we discussed `spec` and `status` as two attribute categories of a Kubernetes object. The `spec` defines the desired status of the Kubernetes object, while `status` defines the actual status of the object. The role of Kubernetes controllers is to maintain the status of the Kubernetes cluster at its desired state. Controllers always keep observing the status of the Kubernetes cluster. For example, when we ask Kubernetes to create a Deployment with five replicas, it's the responsibility of the Deployment Controller to understand that request and create replicas. In section B.18 we discuss this in detail.

B.17 Ingress

Ingress is a Kubernetes object, which routes traffic from outside the cluster to a Kubernetes service over HTTP or HTTPS. It helps exposing one or more Kubernetes Services (say of NodePort type) through a single IP address. In other words, you can think of Ingress as a level of abstraction over one or more services (like a service is an abstraction over one or more pods). The following yaml defines a sample Kubernetes Ingress object.

Listing B.13. The definition of an Ingress object in yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: manning-ingress
spec:
  rules:
  - http:
    paths:
    - path: /orders      #A
      backend:
        serviceName: order-processing-service
        servicePort: 80
    - path: /customers
      backend:
        serviceName: customer-service
        servicePort: 80
```

#A Any request with the /orders context will be routed to the order-processing-service. This is nodePort service listening on port 80 on its internal cluster IP address.

The above yaml defines an ingress object, which routes traffic to two backend Kubernetes services, based on the URL pattern of the request. If the request comes to /orders URL then Kubernetes will route the request to the order-processing-service and if the request comes to /customers URL then to the customer-service. When you create an Ingress resource in Google Kubernetes Engine (GKE), GKE's Ingress controller (section B.16), updates

the GKE load balancer with the corresponding configuration. In fact you cannot just have an Ingress resource, there has to be an Ingress controller too. For example, you can use Nginx as an Ingress controller.

B.18 Kubernetes internal communication

In section B.2 we discussed about Kubernetes master nodes and worker nodes. A master node has its own set of components and worker node has its own too. Let's see how these components communicate with each other. To check the status of the components running in the Kubernetes master node or the control plane, let's use the following kubectl command.

```
\> kubectl get componentstatuses
NAME           STATUS  MESSAGE           ERROR
scheduler      Healthy ok
etcd-0         Healthy {"health": "true"}
etcd-1         Healthy {"health": "true"}
controller-manager Healthy ok
```

When we run the above command using kubectl running in our local machine, it simply talks to an API running on the Kubernetes API server, running within the control plane. The API server internally checks the status of all the control plane components and sends back the response. All kubectl commands we used in this appendix worked in a similar manner.

B.18.1 How kubectl run works?

The best way to understand how Kubernetes internal communication happens is to go through one simple command and see how it works end-to-end (see figure B.4). In section B.6 we used the following kubectl command to create a Kubernetes deployment, with the Docker image: gcr.io/google-samples/hello-app.

```
\> kubectl run hello-server --image gcr.io/google-samples/hello-app:1.0 --port 8080
```

When we run the above command from kubectl running in our local machine, it creates a request with the provided parameters and talks to an API running on the Kubernetes API server. The communication between all the components in the Kubernetes control plane happens via the API server. Following lists out the sequence of events happens in the Kubernetes cluster, once the API server receives a request from kubectl.

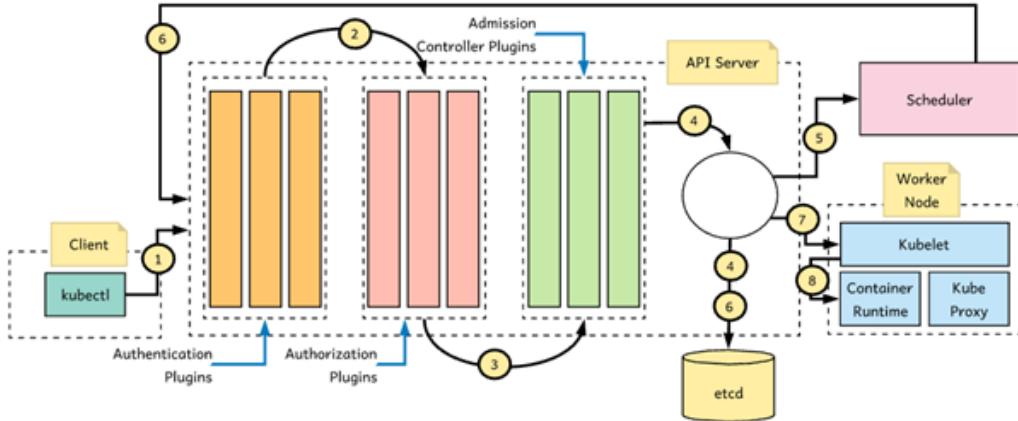


Figure B.4 A request generated by kubectl passes through authentication, authorization and admission controller plugins of the API server, validated and then stored in etcd. Scheduler and kubelet respond to events generated by the API server.

- All the requests coming to the API server are intercepted by an authentication plugin deployed in the API server. This plugin makes sure only the legitimate clients can talk to the API server. The step-1 in figure B.4.
- Once done with authentication, an authorization plugin deployed in the API server intercepts the request and checks whether the authenticated user has permissions to perform the intended action. For example, not all the users are allowed to create pods. The step-2 in figure B.4.
- The authorized API request now goes through a set of Admission Controller plugins. The Admission Controller plugins can perform multiple tasks on the API request. For example, EventRateLimit plugin can limit the number of API requests by user. The step-3 in figure B.4.
- Finally, the API server validates the API request and persists the corresponding objects (corresponding to the API request) in a key-value store, which is the etcd. The etcd is a highly available key-value store, which Kubernetes uses to persist cluster data. API server persists the data related to Kubernetes objects in etcd. The step-4 in figure B.4.
- Once the API server performs any operations on an object, it notifies a set of registered listeners. The scheduler, another component running in the control plane, which has registered with the API server, receives a notification when the API server, as per the API request creates the new pod object in the etcd. The step-5 in figure B.4.
- Scheduler finds a node to run the pod and once again updates the pod definition stored in etcd with node information via the API server. Neither the API server nor the scheduler actually creates the pod. The step-6 in figure B.4.
- Once again the update action performed in step-6, triggers another event. The kubelet

component running in the corresponding worker node (where the pod is supposed to run) picks that event (or gets notified) and starts creating the pod. Each worker node has a kubelet and it keeps listening to the API server to capture any events. The step-7 in figure B.4.

- While creating the pod, kubelet asks the container runtime (for example Docker), which is running in the same worker node to pull the corresponding container images from the registry and start them. At the time of this writing Kubernetes uses Docker as its default container runtime, but with the introduction of container runtime interface (CRI) Kubernetes makes container runtimes pluggable. The step-8 in figure B.4.
- Once all the containers in the pod start running, kubelet keeps monitoring their status and reports to the API server. If it receives an event to terminate the pod, it terminates all the containers in the corresponding pod and updates the API server.

We discussed Kubernetes Controllers in section B.16. While Kubernetes cluster is up and running, the responsibility of these controllers is to make sure the desired state of cluster matches the actual state. For example, the ReplicationSet controller watches the API server to find the status of the pods it controls. If the desired number of pods requested at the time we create the deployment is less than the actual number of pods running in the cluster, it creates the missing number of pods by new pod manifests by talking to the API server. Then it follows the same flow we explained above – the scheduler will pick the pod creation event and will schedule new pods to run on a set of nodes.

B.18.2 How Kubernetes routes a request from an external client to a pod?

We discussed three ways to open up a pod running in a Kubernetes cluster to an external client. Let's summarize that below.

- With a service of NodePort type (section B.3.3)
- With a service of LoadBalancer type (section B.3.3)
- Using an Ingress Controller with a NodePort service type (section B.17)

For the simplicity, let's take the 2nd scenario, assuming that we have a service of LoadBalancer type and see the sequence of events happens in the Kubernetes cluster, when it receives a request from an external client. The following lists out the actions take place when you create a Service in Kubernetes.

- Each worker node in a Kubernetes cluster runs a component called kube-proxy. Whenever Kubernetes creates a Service (either of NodePort type or of LoadBalancer type), kube-proxy gets notified and it opens up the corresponding port (nodePort) in the local node. Each kube-proxy component in each worker node does the same. For the Services of ClusterIP type, there is no nodePort involved.
- A kube-proxy operates in one of the three modes: userspace, iptables and ipvs – we'll keep the ipvs mode outside this discussion.

- If the kube-proxy operates under userspace mode, it installs iptables³ rules on the node, which routes the traffic that comes to the nodePort (for the Services of LoadBalancer type and NodePort type) of the node (via the IP of the node) to the proxy port of the kube-proxy. It also updates iptables rules to route any traffic comes via Cluster IP (for any Service type) on a Service port to the proxy port of kube-proxy. The Cluster IP is a virtual IP address, which Kubernetes creates for all the Services.
- If the kube-proxy operates under iptables mode, it installs an iptables rule, which can route any traffic directed to the nodePort (for the Services of LoadBalancer type and NodePort type) of the node to a randomly picked pod IP address from the Endpoints object corresponding to the Service object – and to the corresponding container port. Also kube-proxy updates the iptables rules to do the same even if it receives traffic for a Service on a Cluster IP on a Service port to a randomly picked pod IP address from the Endpoints object corresponding to the Service object – and to the corresponding container port.

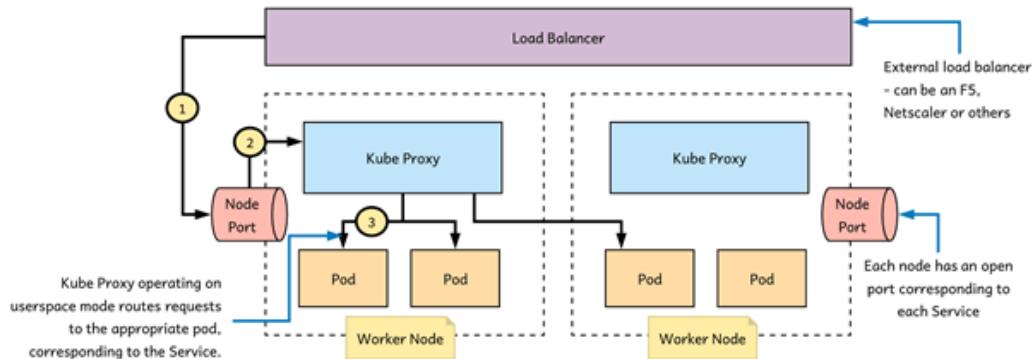


Figure B.5 A kube-proxy operating on userspace mode routes requests to an appropriate pod, corresponding to the Service.

The following lists out the actions take place when you send a request to a Service of LoadBalancer type.

- The client application, which is outside the Kubernetes cluster, sends a request with the IP address pointing to the load balancer. The load balancer runs outside the Kubernetes cluster – and this can be an F5, Netscaler or any other load balancer. When we run the command `kubectl get service` against a Service, the EXTERNAL-IP column represents the IP address of the load balancer. Each service will have its own external IP address on the load balancer. Then again this implementation varies by the cloud vendor. On GKE, it generates a different external IP address for each Service.

³ Iptables is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel: <https://linux.die.net/man/8/iptables>

- Once the request hits the load balancer, and based on the IP address in the request, the load balancer knows the corresponding Service – hence can figure out the nodePort associated with that Service and also the Cluster IP.
- Based on the load-balancing algorithm, load balancer picks a node from the Kubernetes cluster and routes the request to the corresponding nodePort via the corresponding node IP (step-1 in figure B-5).
- If the kube-proxy operates under the userspace mode, then the request routed to the node from the load balancer, will go through the kube-proxy and it will find the corresponding Service (by the nodePort – and each Service has its own nodePort) and the appropriate pod (by default using a round robin algorithm) and reroute the request to that pod (to the pod IP address and the pod port). This pod may even come from a totally different node (steps-2 and 3 in figure B.5). The kube-proxy finds the IP address of the pod, by looking at the Endpoints object attached to the corresponding Service.
- If the kube-proxy operates under the iptables mode, the request won't route through the kube-proxy – but according to the corresponding iptables rules in that node, the request will be rerouted to one of the pods associated with the corresponding service (steps-2 and 3 in figure B.6).
- Once the request hits pod, then it will be dispatched to the corresponding container, by looking at the port. In a given pod, there cannot be multiple containers running on the same port.

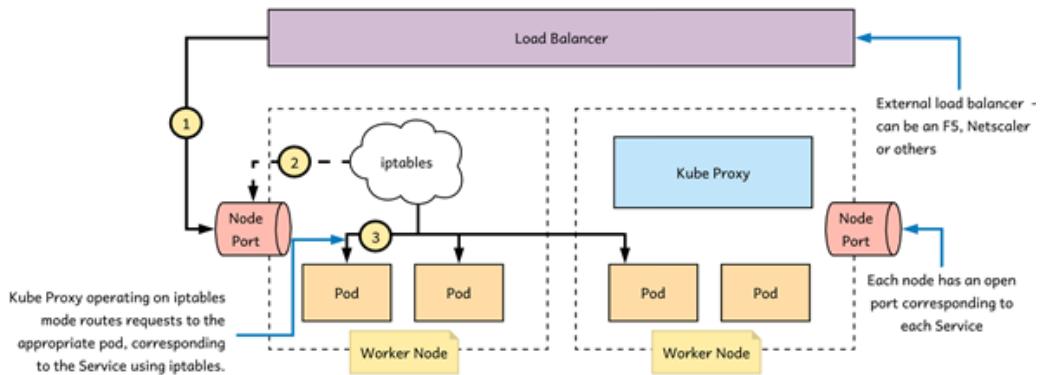


Figure B.6 A kube-proxy operating on iptables mode routes requests to an appropriate pod, corresponding to the Service using iptables.

B.19 Managing configurations

In a typical Kubernetes deployment, some configuration data you use within some containers do change from environment to environment. For example, a pod deployed in the production Kubernetes cluster will have different certificates, database connection details, and so on, from the same pod, which is deployed in a pre-production cluster. First, let's see different ways of

carrying out configuration data in a Kubernetes cluster and then see how to decouple configuration data from the Kubernetes Deployment definition.

B.19.1 Hard-coding configuration data into the Deployment definition

The straightforward way to carry configuration data in a Kubernetes deployment is to hard code those into the definition of the Deployment as environment variables. For example, we can modify the command we used in section B.6 to create a Deployment, in the following way, to pass some configuration data as environment variables in the command line. When you run the command, you will get an error if the deployment already existing. In that case you can use `kubectl delete deployment hello-server` to delete the deployment and re-run the following command.

```
\> kubectl run hello-server --env="name1=value1" --env="name2=value2" --image gcr.io/google-samples/hello-app:1.0 --port 8080
```

Let's run the command in listing B.14 to get the definition of the `hello-server` Deployment in yaml format, and you will notice that the two environment variables we passed in the above command under the `--env` argument are added to the Deployment definition. The actual code or the process corresponding to the `hello-server` container, which runs inside a pod within the `hello-server` Deployment, can read the value of the environment variable. For example, if the server wants to connect to a database, then it can read the database connection details from these environment variables.

Listing B.14. The definition of the hello-server Deployment in yaml

```
\> kubectl get deployment hello-server -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: hello-server
    name: hello-server
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-server
  template:
    metadata:
      labels:
        run: hello-server
    spec:
      containers:
        - env: #A
          - name: name1
            value: value1
          - name: name2
            value: value2
        image: gcr.io/google-samples/hello-app:1.0 #B
        name: hello-server #C
      ports:
        - containerPort: 8080
```

```
protocol: TCP
```

```
#A Lists out all the environment variables passed in the command line
#B The name of the Docker image
#C The name of the container, which runs inside the pod
```

When we create a Deployment in the above way, we couple the configuration data to the Deployment object itself. As we discussed before, in a typical production deployment we do not use imperative commands to create a deployment, rather use declarative object configuration (see section B.13). In case of a declarative configuration model, we maintain the definition of the Deployment object (in this case along with all hard coded environment variables) in a yaml file and then use the command `kubectl apply` to create the Deployment. All the samples we have in chapter 11 follow the declarative configuration model. When we hard code the environment variables into the Deployment definition, we need to duplicate the Deployment definition with different environment values for each production and pre-production environments. Basically need to maintain multiple yaml files for the same Deployment. That is not a good approach – and not recommended.

B.19.2 Introducing ConfigMaps

A ConfigMap is a Kubernetes object, which helps to decouple configuration data from a Deployment. In chapter 11, under section 11.2 we have a comprehensive example of using ConfigMaps. In this section, we discuss different ways of using ConfigMaps in a kubernetes Deployment. If we take the same example, we had in section B.19.2; we can define a ConfigMap to carry the configuration data as in listing B.15.

Listing B.15. The definition of a ConfigMap object that carries text data

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hello-server-cm    #A
data:    #B
  name1: value1
  name2: value2
```

```
#A The name of the ConfigMap
#B Lists out data as name/value pairs and the value is treated as text
```

The listing B.15 creates a ConfigMap object with a text representation. The listing B.16 shows how to create a ConfigMap with binary data. Here the value of the key (`image1`) must be base64 encoded.

Listing B.16. The definition of a ConfigMap object that carries binary data

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hello-server-cm
binaryData:    #A
```

```
image1: /u3+7QAAAIAAAABAAAAAQAGand0..
```

#A Lists out binary data as name/value pairs and the value is in base64-encoded format

To create a ConfigMap object in Kubernetes following the declarative configuration model, we use the following command, assuming that `hello-server-cm.yaml` file carries the complete definition of the ConfigMap.

```
\> kubectl apply -f hello-server-cm.yaml
```

B.19.3 Consuming ConfigMaps from a Kubernetes Deployment and populate environment variables

In this section we discuss, how to consume the configuration data defined in a ConfigMap object from a Kubernetes Deployment and populate a set of environment variables. In other words, the Kubernetes Deployment reads the configuration data from a ConfigMap, and updates a set of environment variables. In listing B.17 you can find the updated `hello-server` deployment and the code annotations explain how it works.

Listing B.17. The definition of the hello-server Deployment in yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: hello-server
  name: hello-server
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-server
  template:
    metadata:
      labels:
        run: hello-server
    spec:
      containers:
        env:
          - name: name1      #A
            valueFrom:
              configMapKeyRef:    #B
                name: hello-server-cm   #C
                key: name1      #D
          - name: name2
            valueFrom:
              configMapKeyRef:
                name: hello-server-cm
                key: name2
        image: gcr.io/google-samples/hello-app:1.0
        name: hello-server
        ports:
          - containerPort: 8080
            protocol: TCP
```

```
#A The name of the environment variable. The corresponding container can read the value of the environment variable
using this name as the key.
#B Instructs Kubernetes to look for ConfigMap object to read the value of the environment variable.
#C The name of the ConfigMap object.
#D The name of the key defined in the ConfigMap object.
```

With this approach we have completely decoupled the configuration data from our Kubernetes Deployment. You can have one single Deployment definition – and have a different ConfigMap object for each production and pre-production environments with different values

B.19.4 Consuming ConfigMaps from a Kubernetes Deployment with volume mounts

In this section we discuss how to read a configuration file from a ConfigMap object and mount that file to a container file system from a Deployment. The listing B.18 shows how we can represent a configuration file in a ConfigMap.

Listing B.18. The definition of a ConfigMap object that carries a config file

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: properties-file-cm
data:
  application.properties: |    #A
  [
    server.port: 8443
    server.ssl.key-store: /opt/keystore.jks
    server.ssl.key-store-password: ${KEYSTORE_SECRET}
    server.ssl.keyAlias: spring
    spring.security.oauth.jwt: true
    spring.security.oauth.jwt.keystore.password: ${JWT_KEYSTORE_SECRET}
    spring.security.oauth.jwt.keystore.alias: jwtkey
    spring.security.oauth.jwt.keystore.name: /opt/jwt.jks
  ]
```

#A The name of the ConfigMap entry is application.properties (the file name) and the value is the content of the file.

Let's see how to consume the ConfigMap defined in listing B.18 from the Deployment in listing B.19 and mount it to the corresponding container file system. The code annotations in listing B.19 explain how it works.

Listing B.19. The definition of the hello-server Deployment with volume mounts

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: hello-server
  name: hello-server
spec:
  replicas: 1
  selector:
    matchLabels:
      run: hello-server
```

```

template:
  metadata:
    labels:
      run: hello-server
  spec:
    containers:
      volumeMounts:    #A
        - name: application-properties    #B
          mountPath: "/opt/application.properties"    #C
          subPath: "application.properties"    #D
      image: gcr.io/google-samples/hello-app:1.0
      name: hello-server
      ports:
        - containerPort: 8080
          protocol: TCP
          volumes:    #E
        - name: application-properties    #F
      configMap:
        name: properties-file-cm    #G

```

#A This section defined properties corresponding to each volume mount.

#B The name of the volume. This refers to the volumes/name element, towards the end of this configuration file.

#C The location in the container file system – or where to mount the file.

#D A sub path from the mountPath, so the root of the mountPath can be shared between multiple volumeMounts. For example if we do not define a subPath here, when have another volumeMount under the opt directory, it would create issues.

#E This section defines a set of volumes, which are referred by name from the containers/volumeMounts section.

#F The name of the volume.

#G The name of the ConfigMap object from listing B.18.

C

Service Mesh and Istio fundamentals

One key aspect of microservices architecture is the Single Responsibility Principle (https://en.wikipedia.org/wiki/Single_responsibility_principle), under which a microservice should be performing only one particular function. In chapter 3 we discussed how to use the API Gateway pattern to take most of the burden out of microservices and delegate security processing at the edge to an API gateway. The API gateway works mostly with north/south traffic, or in other words, the traffic between applications (or consumers) and APIs. But still, in the examples we discussed in chapter 6 and chapter 7, most of the processing while securing inter-microservice communications (or east/west traffic) with mutual Transport Layer Security (mTLS) and JSON Web Token (JWT), was carried out by microservices themselves.

The Service Mesh is an architectural pattern with multiple implementations. It deals with east/west traffic (or, in other words, the traffic between microservices) to take most of the burden out of microservices with respect to security processing and other non-functional requirements. A service mesh brings in the best practices in resiliency, security, observability, and routing control to your microservices deployment, which we discuss in detail in the rest of this appendix. This appendix lays you the foundation to follow chapter 12, which focuses on securing a microservices deployment with the Istio service mesh.

C.1 Why services mesh?

A service mesh is a result of the natural progress in implementing Single Responsibility Principle in microservices architecture. If you look at a framework like Spring Boot, it tries to implement some of the key functionalities we see in a typical service mesh today, at the language level as a library. As a microservices developer you do not need to worry about implementing those, but simply use the corresponding libraries in your microservice

implementation. This is the approach we followed in chapter 6 and chapter 7 while securing inter-microservice communications with mutual mTLS and JWT. You can even call this model, an embedded service mesh (figure C.1).

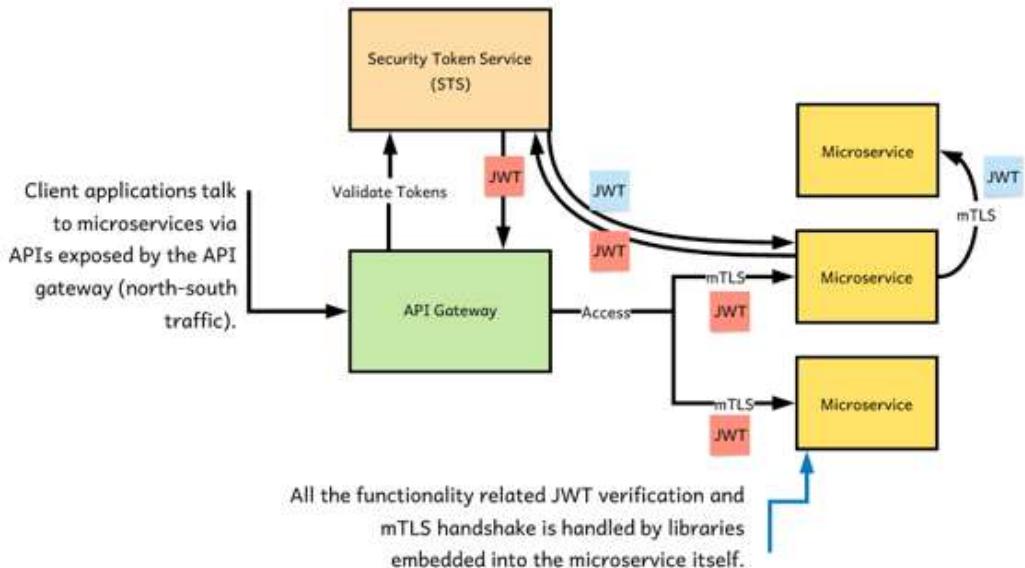


Figure C.1 Embedded service mesh, where each microservice by itself does security processing with a set of embedded libraries.

The embedded service mesh approach has several drawbacks. For example, if you want to use Spring Boot libraries, which implement security, observability and resiliency features, then your microservices implementation must be in Java. Also, if you discover any issues with Spring Boot libraries then you need to redeploy your entire microservice. At the same time whenever there is an API change in Spring Boot libraries, you need to make the corresponding changes in your microservices implementation. All in all, the embedded service mesh approach does not quite fit well into the microservice architecture. In the rest of the appendix, when we talk about the service mesh, it is the out-of-process service mesh. In contrast to the embedded service mesh approach, out-of-process service mesh (figure C.2) runs apart from your microservice and transparently intercepts all the traffic coming into and going out from your microservice.

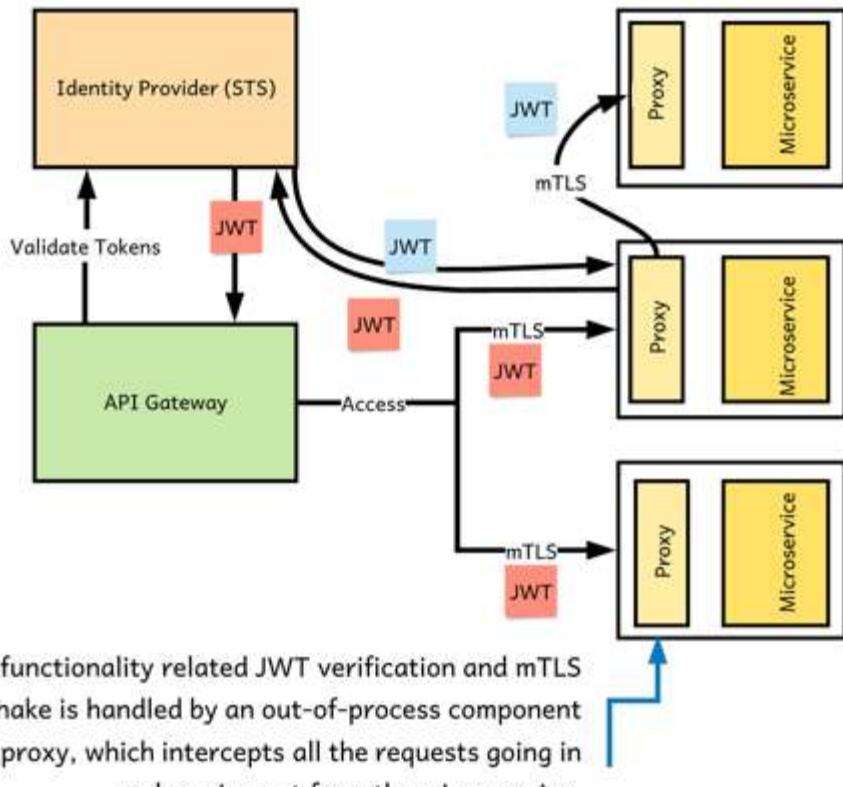


Figure C.2 The out-of-process service mesh, which does security processing via a proxy by intercepting all the requests going in and coming out from the corresponding microservice.

C.2 The natural evolution of microservices deployments

In practice microservices deployments are at different maturity levels. Some just run their services with an embedded lightweight application server (for example Spring Boot) on a physical machine or a virtual machine. This approach is okay, if you have just a few microservices and only a few people working on them.

When the number of services increases, and more teams start working on them, it becomes harder to live without automation. This is the time people start worrying about deploying microservices in containers. It is common for those who are getting started with containers to begin without a container orchestration framework like Kubernetes. This is now changing with more and more cloud providers starting to provide Kubernetes as a service. When the number of containers increases, the management of those becomes a nightmare unless you have a container orchestration framework (see appendix B). This is the time people start worrying

about deploying microservices in containers with a container orchestration framework like Kubernetes.

The next natural step since moving to Kubernetes is to use a service mesh. Kubernetes helps managing a large-scale microservices deployment, but it lacks in providing application level quality of service features for microservices. This is where the service mesh comes in. In other words, Kubernetes is like an operating system, where the service mesh (a product that implements the Service Mesh pattern) runs on top of it to bring in a set of quality of service features for microservices at large scale.

C.2.1 The service mesh architecture

The service mesh architecture primarily consists of two planes: a data plane and a control plane (figure C.3). These two planes coordinate with each other to bring in the best practices in resiliency, security, observability, and routing control to your microservices deployment. Let's first have a look at the data plane.

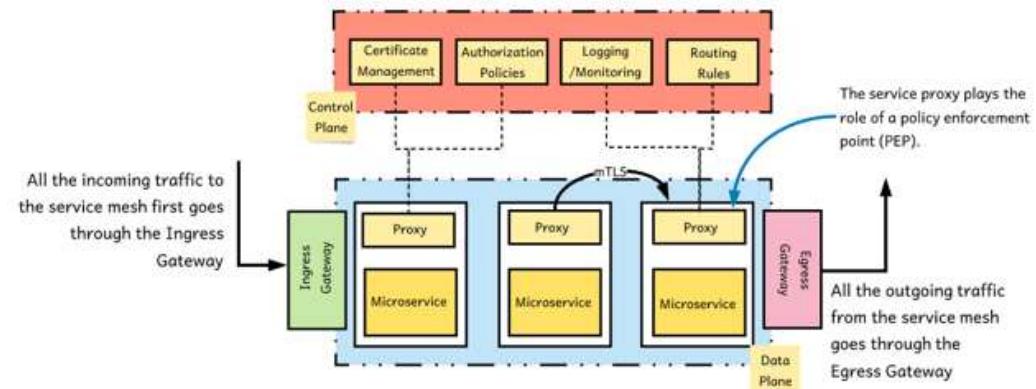


Figure C.3 A typical service mesh consists of a control plane and a data plane.

DATA PLANE

In our Spring Boot examples (in chapters 6 and 7), where we implemented an embedded service mesh with Spring Boot libraries, a Spring Boot interceptor intercepts all the requests coming to your microservice. In the same way, in an out-of-process service mesh implementation, we have a proxy that intercepts all the requests coming in and going out of your microservice. We call this a *service proxy* (figure C.3). Since the service proxy is in the request/response path, it can centrally enforce security; monitor; manage traffic; perform service discovery; and implement patterns to support resiliency, such as circuit breaker, bulk head and so on. In other words, the service proxy plays the role of a policy enforcement point (PEP).

NOTE In this book we only focus on security in a service mesh. If you would like to read in detail about other features of a service mesh such as observability, resiliency, traffic management and so on, we recommend that you check out the books *Istio in Action* (Manning Publications, 2019) by Christian Posta, and *Istio: Up and Running* (O'Reilly Media, 2019) by Lee Calcote and Zack Butche.

In a typical service mesh architecture, each microservice has its own service proxy, and in and out traffic from a microservice flows through the service proxy in a transparent manner. The microservice implementation should not worry about the existence of the service proxy. These service proxies that coordinate traffic in a microservices deployment and act as policy enforcement points build the data plane of service mesh architecture.

In addition to the service proxies, there are two other components in a data plane: an *ingress gateway* and an *egress gateway*. All the traffic enters into the microservices deployment first flows through the ingress gateway, and it decides where (or to which service proxy) to dispatch traffic, and all the traffic leaving the microservices deployment flows through the egress gateway. In other words, all north-south traffic goes through ingress/egress gateways, while all east-west traffic goes through service proxies.

CONTROL PLANE

The control plane in service mesh architecture acts as a policy administration point (PAP). It defines all the control instructions to operate service proxies in the data plane and never touches any data packets at runtime. A typical control plane implementations provide an API or a UI portal or both to perform administrative tasks, and also runs an agent in each service proxy to pass control instructions.

C.2.2 Service mesh implementations

There are multiple implementations of the Service Mesh architectural pattern and some of the popular ones follow. Out of all them, Istio is the most popular and in the book we focus on Istio.

- Istio: An open source service mesh created by Google, Lyft and IBM. Istio uses Envoy developed by Lyft (written in C++) for the service proxy. In this appendix and in chapter 12, we discuss Istio in detail.
- Linkerd: A service mesh developed by Buoyant and has both an open source version and commercial licenses. Linkerd has its own service proxy written in Rust. You can find more about Linkerd architecture from here: <https://linkerd.io/2/reference/architecture>.
- HashiCorp Consul: A service mesh developed by HashiCorp and has both an open source version and commercial licenses. A new feature called Connect, introduced since HashiCorp Consul 1.2 turned Consul into a service mesh. You can read more about HashiCorp Consul architecture from here <https://www.hashicorp.com/resources/service-mesh-microservices-networking>.
- Aspen Mesh: A commercial service mesh implementation based on Istio. More details about Aspen Mesh are available here: <https://aspenmesh.io/feature-enterprise-service->

mesh.

- AWS App Mesh: A service mesh developed by Amazon Web Services (AWS). You can read more about AWS App Mesh from here: <https://aws.amazon.com/app-mesh>.
- Microsoft Azure Service Fabric: A service mesh implementation by Microsoft on Azure. You can read more about Azure Service Fabric from here: <https://docs.microsoft.com/en-us/azure/service-fabric-mesh/service-fabric-mesh-overview>.
- AVI Networks: A service mesh implementation by AVI Networks, based on Istio. VMware acquired AVI Networks in June 2019. You can find more about AVI Networks service mesh implementation from here: <https://avinetworks.com/universal-service-mesh>.
- Red Hat OpenShift Service Mesh: A service mesh implementation by Red Hat on OpenShift, based on Istio. You can read more about Red Hat OpenShift Service Mesh from here: <https://www.openshift.com/learn/topics/service-mesh>.

Even though there are many service mesh implementations based on Istio, when you dig deep into the details, you will find some differences. For example, this explains the differences between the upstream Istio project and the Red Hat OpenShift Service Mesh: https://docs.openshift.com/container-platform/4.2/service_mesh/service_mesh_arch/ossm-vs-community.html. Also if you are interested in learning the differences between Istio, Linkerd and Consul, here is a good reference: <https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/>. All these service mesh implementations are increasingly evolving projects, so you may need to look for the up to date information all the time.

C.2.3 Service mesh vs. API gateway

In chapter 3, we discussed the role of an API gateway in a microservices deployment. Mostly the API gateway handles north/south traffic – or in other words, the communication between the client applications and the APIs. In contrast to that, in a typical microservices deployment the service mesh predominantly handles east/west traffic, or the communication between microservices. Then again, we also see some service mesh implementations are evolving into handle north/south traffic as well, where some components in the service mesh also plays the role of an API gateway.

C.3 Istio service mesh

Istio is a service mesh implementation developed by Google, Lyft and IBM. It is open source and the most popular service mesh implementation at the time of this writing. The project started in May 2017, using Envoy as the service proxy, which runs in the data plane. The control plane components are developed in Go programming language. The Istio code base is available on GitHub at <https://github.com/istio/istio>. One of the key metrics to find the popularity and the adoption of an open source project is the number of GitHub stars and at the time of this writing (Nov, 2019) Istio has almost 20,000 stars. In this appendix, we explain

Istio to an extent to lay the foundation for chapter 12, where we discuss securing microservices in an Istio environment.

C.4 Istio architecture

As discussed in section C.3, a typical service implementation operates in two planes: data plane and control plane (figure C.4). In the sections C.4.1 and C.4.2 we discuss how Istio operates in each of those planes.

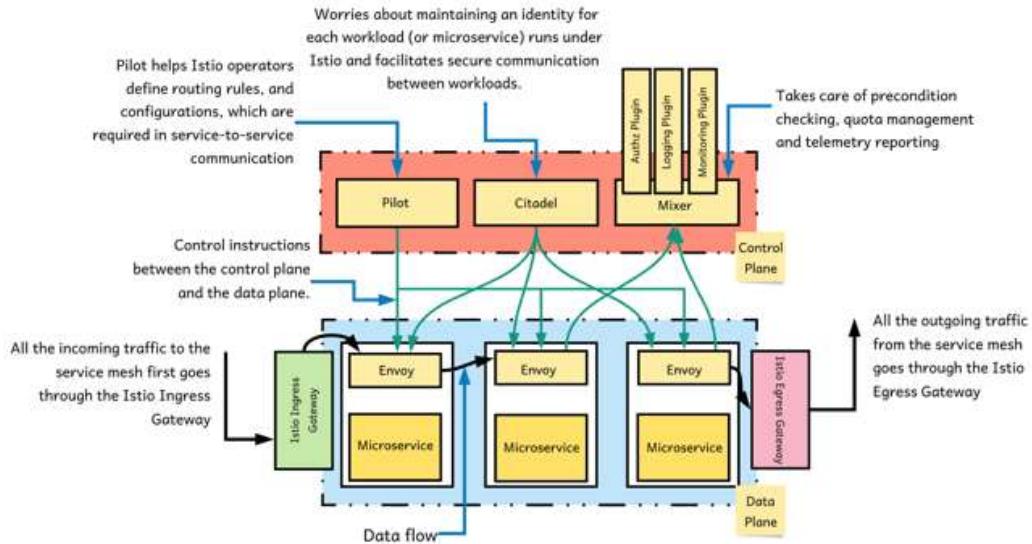


Figure C.4 Istio high-level architecture with a control plane and a data plane.

C.4.1 Istio data plane

The Istio data plane consists of a set of service proxies (along side with each microservice), an ingress gateway and an egress gateway. In the following sections we discuss the responsibilities of each of those components and how they operate in a Kubernetes deployment.

SERVICE PROXY (ENVOY)

Istio out of the box uses Envoy as its service proxy. In a typical Kubernetes deployment, Envoy is deployed in each pod as a sidecar¹ along with the corresponding microservice. Kubernetes makes sure all the containers in a given pod, run in the same node². Istio also

¹ A sidecar is a container that runs in the same pod along with the container that runs the microservice. In a typical pod, there can be one main car (which runs the microservice) and multiple sidecars. A sidecar can act as proxy to the main car, or as a container that provides some utility functions.

² A Kubernetes node can be a physical machine or a virtual machine. A node runs multiple pods.

updates the iptables³ rules in the corresponding Kubernetes node to make sure, all the traffic comes to the container that carries the microservice first flows through Envoy – and in the same way any traffic initiated from the microservice also flows through Envoy. That way Envoy takes control of all the traffic going in and coming out of the microservice. Following lists out the core functionalities Envoy supports as a service proxy.

- HTTP/2 and gRPC support

Envoy supports for HTTP/2 and gRPC for both incoming and outgoing connections. In fact Envoy was one of the very first HTTP/2 implementations. gRPC (<https://grpc.io/>) is an open source remote procedure call framework (or a library), originally developed by Google. It's in fact the next generation of a system called Stubby, which Google has been using internally within Google for over a decade. gRPC achieves efficiency for communication between systems using HTTP/2 as the transport and protocol buffers as the interface definition language (IDL). We discuss gRPC in detail in appendix J. HTTP/2 provides request multiplexing and header compression, which increases its performance significantly with compared to HTTP/1.1. It also employs binary encoding of frames, which makes the data being transferred much more compact and efficient for processing. You can read more about HTTP/2 in appendix J or check the book, *HTTP/2 in Action* (Manning Publications, 2019) by Barry Pollard.

- Protocol Translation

Envoy is also an HTTP/1.1 to HTTP/2 transparent proxy. In other words, it can accept an HTTP/1.1 request and proxy it over HTTP/2. You can also send JSON payload over HTTP/1.1 and Envoy will translate that to a gRPC request over HTTP/2 and send to the corresponding microservice. And further Envoy can translate the gRPC response it gets from the microservice to JSON over HTTP/1.1.

- Load Balancing

The Envoy proxy can act as a load balancer for upstream services. In other words, when one microservice talks to another microservice, that request first goes through the Envoy proxy sitting with the first microservice (figure C.5). This envoy proxy can act as a load balancer for the second microservice, which is called the upstream microservice. Envoy supports advanced load balancing features including automatic retries, circuit breaking, global rate limiting, request shadowing, zone local load balancing, and so on. You can learn more about Envoy load balancing features from the Envoy documentation available at

https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing.html.

³ Iptables is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel: <https://linux.die.net/man/8/iptables>

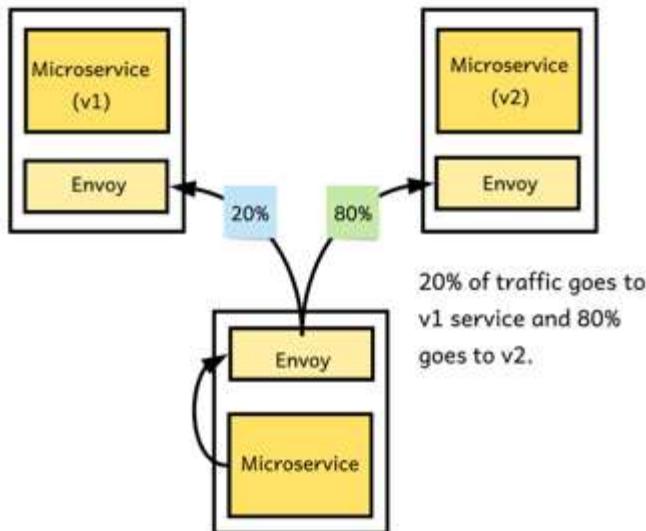


Figure C.5 Envoy does load balancing for upstream services.

- Observability

The four main pillars of observability are metrics, tracing, logging, and visualization. Each of these factors is important to monitor a microservice effectively. In appendix F, we discuss these four pillars and the need for observability in detail. Envoy proxy, which intercepts all the in and out requests from a microservice is in a great position to generate statistics in a transparent manner. It generates stats at three levels: downstream, upstream and server. The downstream stats are related to all incoming connections, while upstream stats are related to all outgoing connections. The server stats are related to health of the Envoy proxy itself, for example CPU, memory usage and so on. Envoy publishes all the stats it collects to Mixer. Mixer is an Istio control plane component, which we discuss in section C.3.3. Envoy does not need to do it for each request rather it can cache the stats and then infrequently push to Mixer.

Unlike in a monolithic deployment, in a typical microservices deployment when a request spans across multiple endpoints, simply logs and stats are not enough. There should be a way to correlate logs between endpoints. When Envoy proxy initiates a request to an upstream service, it generates a unique identifier to trace the request and send it as a header to the upstream service. When the Envoy proxy publishes its downstream stats to the Mixer, it will also publish the corresponding tracing identifiers. Also, if one upstream service wants to talk to another upstream service, it passes through the tracing identifier it got from the first downstream service. With this model, when all the stats generated from Envoy proxies collected centrally, we can build a

complete picture of each request, by correlating tracing identifiers. Under section C.4.2 you will learn more about how Istio handles tracing at the control plane. You can read more about the observability support in Envoy from here https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/observability/observability.

- Security

Envoy acts as a security checkpoint or a policy enforcement point (PEP) for the microservice behind it. One of the emerging patterns we see in the microservices security domain is the zero-trust network pattern. In simple words, this says do not trust the network. If we do not trust the network, then we would need to carry out all the security screening as much as closer to the resource we want to protect – or in our case the microservice. Envoy does that in the service mesh architecture. Envoy intercepts all the requests coming to the microservice it backs, makes sure they are properly authenticated, authorized and then dispatch to the microservice. Since both the Envoy and the microservice run on the same pod, on the same node, the microservice is never exposed outside the node – and also not outside the pod. No request can reach to the microservice, without saying hello to Envoy!

Envoy proxy supports enforcing mutual Transport Layer Security (mTLS), JSON Web Token (JWT) verification, role-based access control (RBAC), and so on. In chapter 12 we discuss in detail all the security features Envoy and Istio support. You can also read more about the security features Envoy supports from here https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/security/security.

INGRESS GATEWAY

Ingress is a Kubernetes resource, which routes traffic from outside the cluster to a Kubernetes service over HTTP or HTTPS. It helps exposing multiple Kubernetes services (say of NodePort type) through a single IP address. In appendix B, we discussed how Ingress works in a Kubernetes cluster. In order for an Ingress resource to work, we also need to have an Ingress controller in place. Some Kubernetes deployments use Nginx, Kong and so on as the Ingress controller. Google Kubernetes Engine (GKE) has its own open source Ingress controller (<https://github.com/kubernetes/ingress-gce>). When you install Istio, it introduces its own Ingress gateway (figure C.6), and updates the iptables rules so that all the traffic enters into the Kubernetes cluster goes through the Istio Ingress gateway, instead of the Kubernetes Ingress controller. The Istio ingress gateway is in fact an Envoy proxy. All the traffic enters into the Kubernetes deployment now flows through Envoy proxy and the Envoy proxy can do monitoring, routing and enforce security.

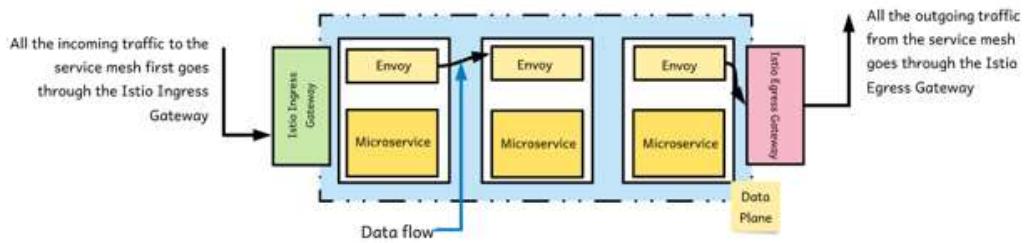


Figure C.6 All the incoming requests to the service mesh go through the ingress gateway and any outbound calls though the egress gateway.

EGRESS GATEWAY

Similar to the Ingress gateway, Istio also introduces an Egress gateway (figure C.6), where all the traffic leaves the Kubernetes deployment goes through Egress gateway. Once again it is an Envoy proxy, which runs as the Egress gateway. For example if your microservice wants to talk an endpoint outside your Kubernetes cluster, then that request will go through the Istio egress gateway – and you can have your own security policies and traffic control rules enforced at the Egress gateway.

C.4.2 Istio control plane

The Istio control plane consists mainly of four components: pilot, galley, mixer and citadel. In the following sections we discuss the responsibilities of each of those components and how they operate in a Kubernetes deployment.

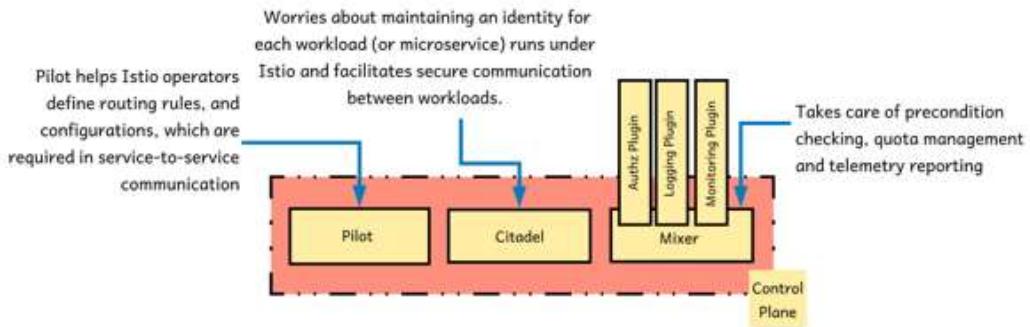


Figure C.7 Istio control plane consists of four components: pilot, galley, mixer and citadel. Galley is not shown in the figure – as its internal to the control plane, which only deals with the underlying infrastructure.

PILOT

Pilot helps you define routing rules, and configurations, which are required in service-to-service communication. For example you can have a routing rule, which says 20% of the traffic

goes to the v1 of the Delivery microservice and 80% goes to the v2. Also you can setup connection timeouts when your service talks to another service, as well as the number of retry attempts. Further you can define parameters with respect to the circuit breaker pattern. When the Order Processing microservice talks to the Delivery microservice, and if the Delivery microservice is down, you can configure a circuit breaker, which will break the circuit between the Order Processing microservice and the Delivery microservice. When the circuit is in open state (or broken), no communication happens between the two microservices, and instead of calling the Delivery microservice, the Order Processing microservice probably could use some pre-configured default values to simulate the response from the Delivery microservice. Then again, after n number of seconds (or minutes), the circuit breaker will try to connect to the Delivery microservice and if it works fine will close the circuit otherwise will keep it open till the next retry.

Pilot exposes an API for Istio administrators (or operators) to define policies and configurations, and another API for Envoy proxies running in the data plane to pull configuration related to them. Once Envoy pulls the related policies and configurations from Pilot, it creates its own Envoy specific configuration file. The listing C.1 shows an example of how Envoy configures a circuit breaker.

Listing C.1. Envoy configuration for circuit breaker

```
"circuit_breakers": {
    "thresholds": [
        {
            "priority": "DEFAULT",
            "max_connections": 100000,
            "max_pending_requests": 100000,
            "max_requests": 100000
        },
        {
            "priority": "HIGH",
            "max_connections": 100000,
            "max_pending_requests": 100000,
            "max_requests": 100000
        }
    ]
}
```

The following is another example of how Envoy keeps connection properties when it wants to connect to upstream microservices.

```
"upstream_connection_options": {
    "tcp_keepalive": {
        "keepalive_time": 300
    }
},
```

GALLEY

To feed policies and configurations into Envoy, Pilot has to interact with the Kubernetes APIs. Galley is a component, which runs in the control plane, which abstracts out the nitty-

gritty details of the platform underneath. For example, when we run Istio on Kubernetes, the Galley knows how to talk to Kubernetes and find the information Pilot needs – and Pilot can work in a platform agnostic way.

MIXER

Mixer, which runs in the control plane, takes care of precondition checking, quota management and telemetry reporting. For example, when a request hits the Envoy proxy at the data plane, it will talk to the Mixer API to do precondition checking to see whether its okay to proceed with that request or not. Mixer has rich plugin architecture, so you can chain multiple plugins in the precondition check phase. For example, you can have a mixer plugin, which can connect to an external policy decision point (PDP) to evaluate a set of access control policies against the incoming request. You can find here a set of available Mixer plugins for Istio: <https://istio.io/docs/reference/config/policy-and-telemetry/adapters/>.

Mixer has two subcomponents in it: policy and telemetry. The policy component enforces polices with respect to precondition checking and quota management. The telemetry component handles functionality related to logging, tracing and metrics. The Envoy proxy from the data plane publishes statistics to Mixer and Mixer can connect to an external monitoring and distributed tracing system like Prometheus, Zipkin, Grafana and so on.

CITADEL

Citadel is the Istio control plane component, which maintains an identity for each workload (or microservice) that runs under Istio and facilitates secure communication between workloads. Citadel provisions X.509 certificates to each workload and manages them. We discuss Citadel in detail in chapter 12.

C.5 Setting up Istio in Google Kubernetes Engine (GKE)

The Istio documentation available at <https://istio.io/docs/setup> explains how to setup Istio in a Kubernetes environment. In this section, we assume you already have access to Google Kubernetes Engine (GKE) and have a Kubernetes cluster up and running. If you need any help setting that up, please check appendix B. In appendix B, we discussed how to create a project in GKE and then a Kubernetes cluster. Run the following `gcloud` command to list out information related to your GKE cluster.

```
\> gcloud container clusters list
NAME          LOCATION    MASTER_VERSION   MASTER_IP      MACHINE_TYPE
manning-ms-security us-west1-a 1.13.7-gke.24  35.203.148.5 n1-standard-1

NODE_VERSION  NUM_NODES  STATUS
1.13.7-gke.8 *     3        RUNNING
```

There are two ways to add Istio support to GKE. Either you can install the open source Istio version by yourself, from the scratch on GKE, or enable Istio as an add-on on GKE. In this appendix we follow the add-on approach. If you would like to install open source Istio from scratch by yourself, please follow the instructions documented here:

<https://cloud.google.com/istio/docs/how-to/installing-oss>. To add Istio support as an add-on for an existing GKE Kubernetes cluster, in our case to the `manning-ms-security` cluster, use the following `gcloud` command. Istio support in GKE is still at the beta level, so we have to use `gcloud beta`, instead of just `gcloud`. At the time you read the book, if the Istio support on GKE has matured to General Availability (GA), you can skip using `beta` in the following commands.

```
\> gcloud beta container clusters update manning-ms-security --update-addons=Istio=ENABLED -  
-istio-config-auth=MTLS_PERMISSIVE
```

Here we use `MTLS_PERMISSIVE` as the Istio auth config, which makes mutual Transport Layer Security (mTLS) optional at each service or to be precise at each Envoy proxy. In chapter 12 we discuss in detail different authentication options Istio provides. In case you want to create a new GKE Kubernetes cluster with Istio support, use the following `gcloud` command⁴.

```
\> gcloud beta container clusters create manning-ms-security --addons=Istio --istio-  
config-auth=MTLS_PERMISSIVE
```

If you have multiple Kubernetes clusters in your GKE environment, and if you want to switch between clusters, use the flowing `gcloud` command, with the cluster name.

```
\> gcloud container clusters get-credentials manning-ms-security
```

The Istio version installed on GKE, depends on the version of the GKE cluster. You can find the GKE version to Istio version mapping from here: <https://cloud.google.com/istio/docs/istio-on-gke/installing>. Also Istio has multiple profiles and each profile defines what Istio features you want to have. When you install Istio as an add-on on GKE you have limited flexibility to pick, which Istio features you want to have, unless those are officially supported by GKE. Even when you install Istio by yourself on GKE, still the recommendation is to use the default Istio profile. You can find here, different Istio profiles and which features are available under each profile: <https://istio.io/docs/setup/additional-setup/config-profiles/>.

C.6 What Istio brings into a Kubernetes cluster?

Once you install Istio on Kubernetes, you'll find a new namespace, a new set of custom resource definitions (CRDs), a set of control plane components as Kubernetes Services and pods, and many others. In this section we discuss the key changes Istio brings into your Kubernetes cluster.

C.6.1 Kubernetes custom resource definitions (CRDs)

A custom resource definition or commonly known as CRD, is a way of extending Kubernetes functionality. A custom resource is in fact an extension to the Kubernetes API. It lets you manage and store custom resources using the Kubernetes API via the Kubernetes API server.

⁴ The list of all available options for `clusters create` command are listed here:
<https://cloud.google.com/sdk/gcloud/reference/beta/container/clusters/create>.

For example Istio introduces a set of custom resources such as, Gateway, VirtualService, ServiceAccount, ServiceAccountBinding, Policy and so on. You can use the command in listing C.2 to list out all the CRDs Istio introduces – and the output here only shows a few of them.

Listing C.2. Some of the custom resource definitions introduced by Istio

```
\> kubectl get crds --all-namespaces | grep istio.io
adapters.config.istio.io           2019-10-27T12:35:43Z
    apikeys.config.istio.io          2019-10-27T12:35:43Z
    attributemanifests.config.istio.io 2019-10-27T12:35:43Z
    authorizations.config.istio.io   2019-10-27T12:35:43Z
    bypasses.config.istio.io         2019-10-27T12:35:43Z
    checknothings.config.istio.io   2019-10-27T12:35:43Z
    circonuses.config.istio.io      2019-10-27T12:35:43Z
    cloudwatches.config.istio.io    2019-10-27T12:35:43Z
```

C.6.2 istio-system namespace

Once you install Istio into your Kubernetes cluster it creates a new namespace called `istio-system`. All the Istio components that run within the control plane (as discussed in C.4.2) are installed under the `istio-system` namespace. The following command lists out all the namespaces in your Kubernetes cluster.

```
\> kubectl get namespaces
NAME        STATUS  AGE
default     Active  27d
istio-system Active  14d
kube-public  Active  27d
kube-system  Active  27d
```

C.6.3 Control plane components

Let's use the command in listing C.3 to list all the Istio components running as Kubernetes Services under the `istio-system` namespace.

Listing C.3. Istio components running as Kubernetes Services

```
\> kubectl get service -n istio-system
NAME            TYPE      CLUSTER-IP      EXTERNAL-IP
istio-citadel   ClusterIP  10.39.240.24   <none>
istio-galley    ClusterIP  10.39.250.154  <none>
istio-ingressgateway LoadBalancer 10.39.247.10  35.230.52.47
istio-pilot     ClusterIP  10.39.243.6   <none>
istio-policy    ClusterIP  10.39.245.132  <none>
istio-sidecar-injector ClusterIP 10.39.244.184  <none>
istio-telemetry  ClusterIP  10.39.251.200  <none>
promsd         ClusterIP  10.39.249.199  <none>
```

Kubernetes exposes all the above Istio components as services. Under section C.4.2 we discussed the responsibilities of `istio-citadel`, `istio-galley` and `istio-pilot`. The `istio-policy` and `istio-telemetry` services are part of mixer. The `istio-ingressgateway` service acts as an ingress gateway, which we discussed under the section C.4.1. The `promsd` service that is based on Prometheus (an open source monitoring system) is

used for metrics. The `istio-sidecar-injector` is used to inject Envoy as a sidecar proxy to Kubernetes pods, which we discuss in detail under the section C.8.1. Something missing in the `istio-system` namespace is the Istio egress gateway (section C.4.1). As we discussed in section C.5, when we install Istio on GKE as an add-on, it only installs the default profile of Istio, and egress gateway is not part of the default profile.

Behind each of the above Service, there is a corresponding pod. The command in listing C.4 lists out all the pods running under the `istio-system` namespace.

Listing C.4. Pods related to Istio running under `istio-system` namespace

| NAME | READY | STATUS | RESTARTS |
|--|-------|-----------|----------|
| istio-citadel-5949896b4b-vlr7n | 1/1 | Running | 0 |
| istio-cleanup-secrets-1.1.12-7vtct | 0/1 | Completed | 0 |
| istio-galley-6c7df96f6-nw9kz | 1/1 | Running | 0 |
| istio-ingressgateway-7b4dcc59c6-6srn8 | 1/1 | Running | 0 |
| istio-init-crd-10-2-2mftw | 0/1 | Completed | 0 |
| istio-init-crd-11-2-f89wz | 0/1 | Completed | 0 |
| istio-pilot-6b459f5669-44r4f | 2/2 | Running | 0 |
| istio-policy-5848d67996-dzfw2 | 2/2 | Running | 0 |
| istio-security-post-install-1.1.12-v2phr | 0/1 | Completed | 0 |
| istio-sidecar-injector-5b5454d777-89ncv | 1/1 | Running | 0 |
| istio-telemetry-6bd4c5bb6d-h5pzm | 2/2 | Running | 0 |
| promsd-76f8d4cff8-nkm6s | 2/2 | Running | 1 |

C.6.4 `istio-ingressgateway` service

Except the `istio-ingressgateway` Service in listing C.4 all the Services are of ClusterIP type. The ClusterIP type is the default service type in Kubernetes and those Services are only accessible within a Kubernetes cluster. The `istio-ingressgateway` Service, which is of LoadBalancer type, is accessible outside the Kubernetes cluster. Let's examine `istio-ingressgateway` service little further, with the following kubectl command.

```
\> kubectl get service istio-ingressgateway -o yaml -n istio-system
```

In the output of the above command you can find the `spec/clusterIP`, which can be used by the Services running inside the Kubernetes cluster to access the `istio-ingressgateway`, and `status/LoadBalancer/ingress/ip` to access the `istio-ingressgateway` from external clients outside the Kubernetes cluster. Also you can notice in the output of the above command an array of ports (under `spec/ports`) with different names (listing C.5). Each element in the ports array represents a different kind of a Service, for example one for HTTP/2 traffic, another for the HTTPS traffic and another for the TCP traffic and so on.

Listing C.5. An array of ports with different names

```
ports:
- name: http2
  nodePort: 31346
  port: 80
  protocol: TCP
```

```

targetPort: 80
- name: https
  nodePort: 31787
  port: 443
  protocol: TCP
  targetPort: 443
- name: tcp
  nodePort: 32668
  port: 31400
  protocol: TCP
  targetPort: 31400

```

Under each element of the ports array you find an element called `name`, `nodePort`, `port`, `targetPort`, and `protocol`. A service of `LoadBalancer` type is also a service of `NodePort` type (see appendix B). Or in other words, the `LoadBalancer` service is an extension to the `NodePort` service and that is why we see a `nodePort` defined under each element in the `ports` array. Each node in the Kubernetes cluster listens on the `nodePort`. For example, if you want to talk to the `istio-ingressgateway` over HTTPS, then you need to pick the value of the `port` element under `name https` – and `istio-ingressgateway` listening on that particular port will reroute the traffic to the corresponding `nodePort` – and then to the corresponding pod, which is listening on `targetPort`. The pod behind the `istio-ingressgateway` runs a container with the Envoy proxy. In section C.10.1 we discuss how to use `istio-ingressgateway` within your microservices deployment – or how to use `istio-ingressgateway` to route requests from external client applications to your microservices.

C.6.5 istio-ingressgateway pod

Lets dive a little deeper into the pod behind the `istio-ingressgateway` service. To find the exact name of the pod, you can use the following command. To filter the pod, here we are using the `--selector` flag, which looks for the provided label in the pod definition. The `istio-ingressgateway` pod carries the label, `istio:ingressgateway`.

```
\> kubectl get pods --selector="istio=ingressgateway" -n istio-system
NAME                      READY   STATUS    RESTARTS   AGE
istio-ingressgateway-7c96766d85-m6ns4   1/1     Running   0          5d22h
```

Now we can log into the `istio-ingressgateway` pod using the following command along with the correct pod name.

```
\> kubectl -it exec istio-ingressgateway-7c96766d85-m6ns4 -n istio-system sh
#
```

In the Envoy file system, if you go inside `/etc/certs` directory you will find the private key (`key.pem`) and the corresponding public cert chain (`cert-chain.pem`) provisioned by Istio Citadel. The ingress gateway uses these keys to authenticate over mutual Transport Layer Security (mTLS) to the upstream service proxies. In chapter 12 we discuss how to enable mTLS between the ingress gateway and the service proxies.

```
# cd /etc/certs
# ls
```

```
# cert-chain.pem key.pem root-cert.pem
```

Also, in the Envoy file system, run the following curl command and you will get the Envoy configurations related to routing and upstream connections.

```
# curl 127.0.0.1:15000/config_dump
```

If you want to save the Envoy configuration to a file in your local machine, then you can run the following command from your local machine. The corresponding Envoy configuration will be written to the `envoy.config.json` file.

```
\> kubectl exec -it istio-ingressgateway-7c96766d85-m6ns4 -n istio-system curl 127.0.0.1:15000/config_dump > envoy.config.json
```

C.6.6 Mesh policy

The `MeshPolicy` is another important thing Istio brings into the Kubernetes cluster. Istio introduces a `MeshPolicy` resource (listing C.6) for all the Services in your Kubernetes cluster across all the namespaces. The `MeshPolicy` resource, which carries the name `default`, defines a cluster-wide authentication policy. It is a must that this global policy should have the name `default`. The `mtls` mode of the policy is set to `PERMISSIVE` – and this is based on the parameters we passed to `gcloud` command to add Istio support to the Kubernetes cluster in section C.5. You can override the global `MeshPolicy` by defining an authentication policy or a set of policies under any namespace. We discuss authentication policies in detail in chapter 12.

Listing C.6. The definition of the MeshPolicy

```
\> kubectl get meshpolicy -o yaml
apiVersion: v1
kind: List
items:
- apiVersion: authentication.istio.io/v1alpha1
  kind: MeshPolicy
  metadata:
    creationTimestamp: "2019-10-14T22:34:06Z"
    generation: 1
  labels:
    app: security
    chart: security
    heritage: Tiller
    release: istio
    name: default
  spec:
    peers:
      - mtls:
          mode: PERMISSIVE
```

C.7 Setting up the Kubernetes deployment

In this section we create two deployments in a Kubernetes cluster under the default namespace. One is for the security token service (STS) and the other one for the Order

Processing microservice. These are exactly the same two services we discussed in chapter 11. In section C.8 we discuss how to engage Istio into these two services.

The source code related to all the samples in this chapter is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, under the appendix-c directory. If you are interested in learning more about the yaml files we use here to create the two deployments for the STS and the Order Processing microservice, please check chapter 11. Run the following command from appendix-c/sample01 directory to create a Kubernetes deployment for STS.

```
\> kubectl apply -f sts.yaml
secret/sts-keystore-secrets configured
deployment.apps/sts-deployment configured
service/sts-service configured
```

Run the following command from appendix-c/sample01 directory to create a Kubernetes deployment for the Order Processing microservice.

```
\> kubectl apply -f order.processing.yaml
configmap/orders-application-properties-config-map configured
configmap/orders-keystore-config-map configured
configmap/orders-truststore-config-map configured
secret/orders-key-credentials configured
deployment.apps/orders-deployment configured
service/orders-service configured
```

Run the following command to list out the deployments available in your Kubernetes cluster under the default namespace.

```
\> kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
orders-deployment   1/1     1           1          2m
sts-deployment    1/1     1           1          2m
```

C.8 Engaging Istio to STS and Order Processing microservice

Engaging Istio to a microservice will result in engaging Envoy with the corresponding microservice at the data plane. Or in other words, we need to inject the Envoy proxy as a sidecar into each pod in our microservices deployment so the Envoy proxy will intercept all the requests and responses to and from the corresponding microservice. There are two ways to inject Envoy as a sidecar proxy. Either you can do it manually by updating your Kubernetes deployment, or you can ask Kubernetes to inject Envoy as a sidecar proxy each time you create a pod in your Kubernetes deployment. The Istio documentation available at <https://istio.io/docs/setup/additional-setup/sidecar-injection> explains the manual process.

C.8.1 Sidecar auto injection

To auto inject Envoy as a sidecar proxy, use the following kubectl command. Here we enable auto injection for the default namespace. The auto inject does not do any magic, it

simply adds a Kubernetes admission controller⁵ to the request path. This new admission controller listens to the API calls to create pods in Kubernetes and modifies the pod definition to add Envoy as a sidecar proxy and also adds another container as an init container. The role of the init container (which we discussed in chapter 11) is to carry out any initialization tasks before any of the containers start functioning in a pod.

```
\> kubectl label namespace default istio-injection=enabled
```

You can use the following `kubectl` command to verify whether `istio-injection` is enabled for the `default` namespace.

```
\> kubectl get namespace -L istio-injection
NAME      STATUS  AGE  ISTIO-INJECTION
default   Active  10h  enabled
istio-system  Active  10h  disabled
kube-public  Active  10h
kube-system  Active  10h
```

When we use auto inject it won't affect any deployment already running unless we restart it. Kubernetes does not have a command to restart a deployment, but we can use the following workaround. Here we first scale down the `orders-deployment` to 0 replicas – so Kubernetes will kill all the pods running in that deployment, and then in the second command we scale up for 1 replica (or any number of replicas you wish). We need to repeat the same for the `sts-deployment` as well.

```
\> kubectl scale deployment orders-deployment --replicas=0
\> kubectl scale deployment orders-deployment --replicas=1
\> kubectl scale deployment sts-deployment --replicas=0
\> kubectl scale deployment sts-deployment --replicas=1
```

Now we can use the following command to find the pod names associated with the `order-deployment` and `sts-deployment` deployments and see what changes Istio has brought into those pods. Looking at the output here we can see there are two containers running inside each pod (`Ready: 2/2`). One container carries the microservice (either the Order Processing microservice or the STS), while the other container is the Envoy proxy.

```
\> kubectl get pods
NAME          READY  STATUS    RESTARTS  AGE
orders-deployment-6d6cd77c6-fc8d5  2/2    Running  0          71m
sts-deployment-c58f674d7-2bspc   2/2    Running  0          72m
```

Let's use the following command to find more about the `orders-deployment-6d6cd77c6-fc8d5` pod.

```
\> kubectl describe pod orders-deployment-6d6cd77c6-fc8d5
```

⁵An admission controller intercepts all the requests coming to the Kubernetes API server.

The above command will result in a lengthy output, however lets only pick independent sections to understand the changes Istio integration has made. We discuss those changes in section C.8.2 and C.8.3.

To disengage Istio from a given Kubernetes namespace, we can use the following kubectl command. Please note that, the hyphen (-) at the end of the command is not a typo.

```
\> kubectl label namespace default istio-injection-
```

C.8.2 Setting up iptables rules

When you look at the pod description of `orders-deployment-6d6cd77c6-fc8d5` (from section C.8.1) you will notice the following section, which defines an init container, with `proxy_init` Docker image. As we discussed before, an init container runs before any other container in the pod. The responsibility of `proxy_init` is to update the `iptables`⁶ rules of the pod, so that any traffic comes in and goes out of the pod, will go through Envoy proxy.

```
Init Containers:
  istio-init:
    Container ID: docker://54a046e5697ac44bd82e27b7974f9735
    Image:         gke.gcr.io/istio/proxy_init:1.1.13-gke.0
```

C.8.3 Envoy sidecar proxy

How Envoy proxy is added into the pod as a container is another important section you find in the pod description of `orders-deployment-6d6cd77c6-fc8d5`. This container listens on port 15090, while the container, which carries the Order Processing microservice listens on port 8443.

```
Containers:
  istio-proxy:
    Container ID: docker://f9e19d8248a86304d1a3923689a874da0e8fc8
    Image:         gke.gcr.io/istio/proxyv2:1.1.13-gke.0
    Image ID:      docker-pullable://gke.gcr.io/istio/proxyv2@sha256:829a7810
    Port:          15090/TCP
    Host Port:    0/TCP
```

The way client applications reach the Order Processing microservice running in the `orders-deployment-6d6cd77c6-fc8d5` pod is via a Kubernetes Service (which we discussed in chapter 11). You can use the following kubectl command to describe the `orders-service` Service (we are still not using an Istio Gateway, which we discuss in section C.10.1), and in the output you find that the Service redirects traffic to port 8443 of the `orders-deployment-6d6cd77c6-fc8d5` pod. What `proxy_init` init container does is (as

⁶ `Iptables` is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel: <https://linux.die.net/man/8/iptables>

discussed in section C.11.2), it updates the `iptables` rules so that any traffic comes to port 8443 will transparently redirected to port 15090, where Envoy proxy listens to.

```
\> kubectl describe service orders-service
Name:           orders-service
Namespace:      default
Labels:         <none>
Selector:       app=orders
Type:          LoadBalancer
IP:            10.39.249.66
LoadBalancer Ingress: 35.247.11.161
Port:          <unset>  443/TCP
TargetPort:    8443/TCP
NodePort:      <unset>  32401/TCP
Endpoints:     10.36.2.119:8443
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>
```

C.9 Running the end-to-end sample

In this section we test the end-to-end flow (figure C.8). First we need to get a token from the security token service (STS), and then use it to access the Order Processing microservice. Now we've got both these services running on Kubernetes, with Istio. Let's use the following `kubectl` command to find the external IP addresses of both the STS and Order Processing microservice.

```
\> kubectl get services
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   ClusterIP  10.39.240.1  <none>        443/TCP      10h
orders-service  LoadBalancer  10.39.242.155  35.247.42.140  443:30326/TCP  9h
sts-service    LoadBalancer  10.39.245.113  34.82.177.76   443:32375/TCP  9h
```

Let's use the following cURL command, run from your local machine, to a get a token from the STS. Make sure you use the correct external IP address of the STS.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" https://34.82.177.76/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works fine, the security token service returns an OAuth 2.0 access token (`access_token`), which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1NTEzMTIzMzYsInVzZXJfbmFtZSI6InBldGVyIiwiXXV0aG9yaXRpZXMiolsiuk9MRV9VVU0VSI1osImp0aSISijRkMmjNj04LTQ2MWQtNGV1Y1hZT1jLTlV1YWUxZjA4ZTJhMiIsImNsawVudef9pZC16ImFwcGxpY2F0aW9uaWQiLCjzY29wZSI6NyJmb28iXX0.tr4yu mGLtsH7q9Ge2i7gxyTs00a0RS0Yoc2uBuAW50VIKZcVsIITWV3bDN0FVHBzimpAPy33tvicFR0hBFoVThqKXzzG00SkURN5bnQ4uFLAP0NpZ6BuDjvvVmwxNxrQp2lVX141Q4eTvuyZozjUSCxzCI1LNw5EFFi22J73g1_mRm2j- dEhbP1TVMaRKLBdk2hzIDVKzu5oj_gODBFm3a1S- IJjYoCimIm2igcesXkipRjtNcrJSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97jlT6Rr1XZB1veyjfKs_XIXgU5qizRm9mt5xg", "token_type": "bearer", "refresh_token": "", "expires_in": 5999, "scope": "foo", "jti": "4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

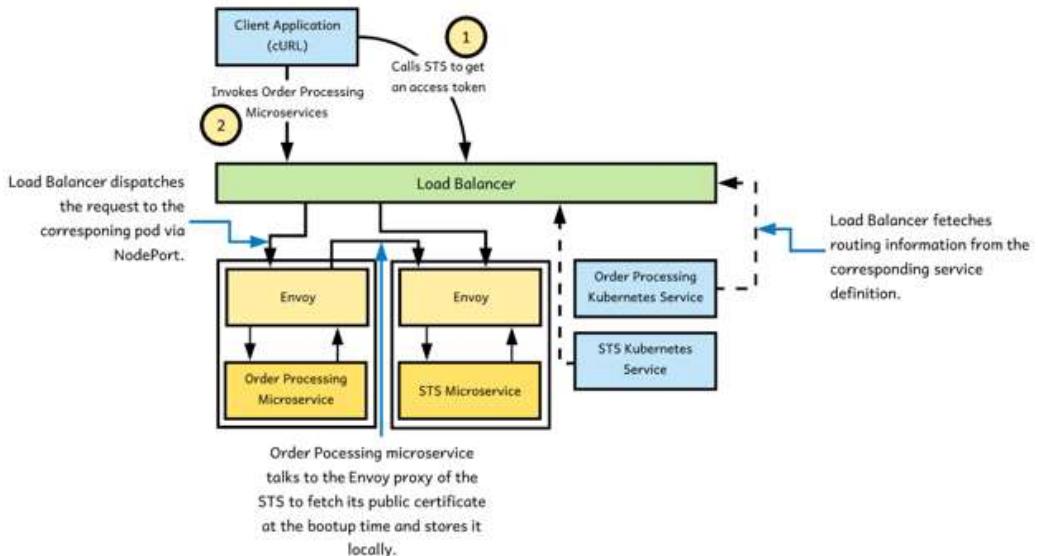


Figure C.8 The client application first talks to the STS to get an access token, and with that token talks to the Order Processing microservice.

Now try to invoke the Order Processing microservice with the JWT you got from the above curl command. Set the same JWT we got in the HTTP Authorization Bearer header using the following curl command and invoke the Order Processing microservice. Because the JWT is a little lengthy, you can use a small trick when using the curl command. First, export the JWT to an environmental variable (`TOKEN`). Then use that environmental variable in your request to the Order Processing microservice.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" https://35.247.42.140/orders/11
{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}
```

The way we implemented the Order Processing microservice does not get the full benefits of the Istio service mesh. For example, we do the mutual Transport Layer (mTLS) validation as well as the JWT validation at the microservice itself, using Spring Boot libraries. Rather we can delegate those tasks to the service mesh itself. We discuss how to do that in chapter 12.

C.10 Updating Order Processing microservice with Istio configuration

In this section we update the Kubernetes deployment we created in section C.7 with respect to the Order Processing microservice and STS microservice with Istio specific configurations. The only thing Istio does so far by engaging with the Order Processing microservice is to have

the Envoy proxy to intercept all the requests coming to and going out from the microservice (figure C.8). When the client application sends a request to the Order Processing microservice it first hits the Kubernetes ingress controller and then the Envoy proxy, which sits with the microservice. Instead of going through the Kubernetes ingress controller or the external load balancer, we want all the requests to the microservice to flow through the Istio ingress gateway (section C.9.4) first (figure C.9).

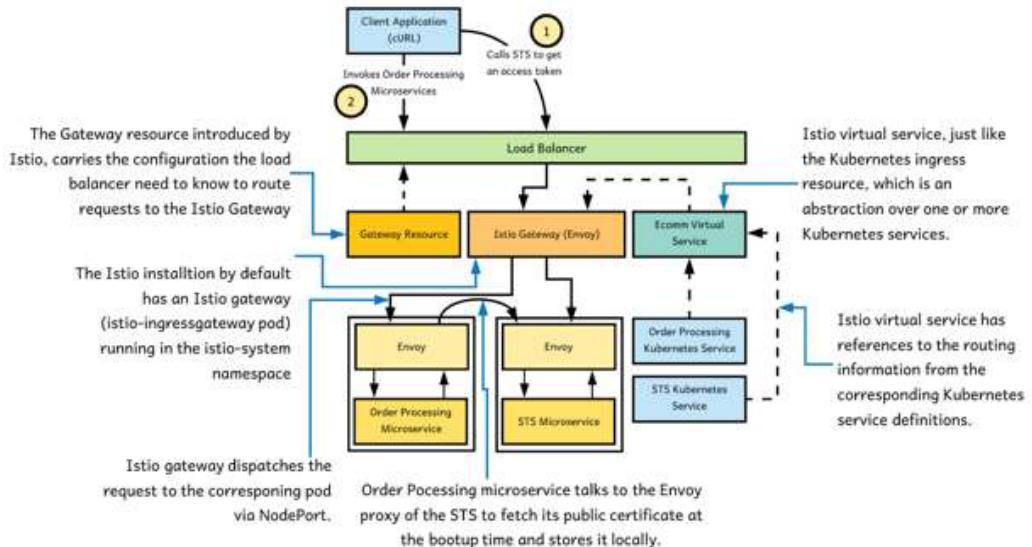


Figure C.9 The Istio gateway intercepts all the traffic coming into Order Processing microservice and the STS microservice.

C.10.1 Creating a Gateway resource

The Gateway resource introduced by Istio, instructs the load balancer on how to route traffic to the Istio ingress gateway. As discussed in section C.6.4, when you install Istio on your Kubernetes cluster it adds `istio-ingressgateway` service and `istio-ingressgateway` pod behind it under the `istio-system` namespace. The `istio-ingressgateway` pod runs an Envoy proxy, and carries the label `istio:ingressgateway`. The listing C.7 shows the definition of the `ecomm-gateway` (a Gateway resource), which instructs the load balancer to route any HTTPS traffic comes to port 443 to the `istio-ingressgateway` pod.

Listing C.7. The definition of the ecomm-gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ecomm-gateway
  namespace: istio-system #A
```

```

spec:
  selector:
    istio: ingressgateway      #B
  servers:
  - port:
    number: 443
    name: http
    protocol: HTTPS
  tls:
    mode: PASSTHROUGH      #C
  hosts:
  - "*"        #D

```

#A Since the ingress gateway is running in the `istio-system` namespace, we also create the `Gateway` resource in the same namespace.

#B This binds the `Gateway` resource to the `istio-ingressgateway` pod, which is an Envoy proxy that carries the label `istio:ingressgateway`.

#C This instructs the Envoy proxy, which runs as the Istio gateway to not to terminate transport layer security (TLS), but just passes it through.

#D Instructs the load balancer to route all the traffic coming to any host on port 443 to the Istio gateway.

To create the `Gateway` resource and also the corresponding `VirtualService` resource (section C.10.2) run the following command from `appendix-c/sample01` directory.

```
\> kubectl apply -f istio.ingress.gateway.yaml gateway.networking.istio.io/ecomm-gateway
      created
virtualservice.networking.istio.io/ecomm-virtual-service created
```

You can use the following `kubectl` command to list out all the `VirtualService` resources available in your Kubernetes cluster under the default namespace.

```
\> kubectl get virtualservices
NAME          GATEWAYS      HOSTS   AGE
ecomm-virtual-service [ecomm-gateway] [*]  6m
```

C.10.2 Creating a VirtualService resource for the Order Processing microservice and STS microservice

The `VirtualService` resource introduced by Istio, instructs the corresponding Istio gateway on how to route traffic to the corresponding Kubernetes service. A Kubernetes service (which we discussed in detail in appendix B) is an abstraction over one or more pods, while an Istio `VirtualService` is an abstraction over one or more Kubernetes services. It is quite similar to the Kubernetes Ingress resource (appendix B) and the `Gateway` resource, which we discussed in section C.10.1, is quite similar to the Kubernetes Ingress controller. The listing C.9 shows the definition of the `ecomm-virtual-service` (a `VirtualService` resource), which instructs the Istio gateway (which we created in section C.10.1) to route any HTTPS traffic that comes on port 443 with the `sni_hosts`⁷ `sts.ecomm.com` to the `sts-service` (a Kubernetes service) and with `sni_hosts` `orders.ecomm.com` to the `orders-service` (a Kubernetes service).

⁷ Server Name Indication (SNI) is a TLS extension, a client application can use before the start of the TLS handshake to indicate the server, which host name it intends to talk to. Istio gateway can route traffic looking at this SNI parameter.

Listing C.9. The definition of ecomm-virtual-service

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ecomm-virtual-service
spec:
  hosts:
  - "*"
  gateways:
  - ecomm-gateway.istio-system.svc.cluster.local
  tls:
    - match:
      - port: 443
        sni_hosts:
        - sts.ecomm.com    #A
      route:
        - destination:
          host: sts-service   #B
          port:
            number: 443     #C
    - match:
      - port: 443
        sni_hosts:
        - orders.ecomm.com   #D
      route:
        - destination:
          host: orders-service   #E
          port:
            number: 443     #F

```

#A If the SNI header carries sts.ecomm.com value, the Istio gateway routes the traffic to sts-service.

#B The name of the Kubernetes service corresponding to the STS pod.

#C The sts-service listens on port 443.

#D If the SNI header carries orders.ecomm.com value, the Istio gateway routes the traffic to orders-service.

#E The name of the Kubernetes service corresponding to the pod, which carries the Order Processing microservice.

#F The orders-service listens on port 443.

C.10.3 Running the end-to-end flow

In this section we test the end-to-end flow (figure C.9). First we need to get a token from the security token service (STS), and then use it to access the Order Processing microservice. Now we've got both these services fronted by the Istio ingress gateway. Let's use the following two commands to find the external IP address and the HTTPS port of the Istio ingress gateway, which runs under the `istio-system` namespace. The first command finds the external IP address of the `istio-ingressgateway` service and exports it to the `INGRESS_HOST` environment variable and the second command finds the HTTPS port of the `istio-ingressgateway` service and exports it to the `INGRESS_HTTPS_PORT` environment variable.

```

\> export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o
  jsonpath='{.status.loadBalancer.ingress[0].ip}')

\> export INGRESS_HTTPS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o
  jsonpath='{.spec.ports[?(@.name=="https")].port}')

```

You can use the following echo command to make sure that we captured the right values for the two environment variables.

```
\> echo $INGRESS_HOST
34.83.117.171
\> echo $INGRESS_HTTPS_PORT
443
```

Let's use the following cURL command, run from your local machine, to get a token from the STS. Here we use the environment variables, which we defined before for the hostname and the port of the `istio-ingressgateway` service. Since we are using Server Name Indication (SNI) based routing at the Istio gateway, we are using the host name, `sts.ecomm.com` to access the STS. Since there is no DNS mapping to this hostname, we are using the `--resolve` parameter in cURL to define the hostname to IP mapping.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" --resolve sts.ecomm.com:$INGRESS_HTTPS_PORT:$INGRESS_HOST https://sts.ecomm.com:$INGRESS_HTTPS_PORT/oauth/token
```

In this command, `applicationid` is the client ID of the web application, and `applicationsecret` is the client secret. If everything works fine, the security token service returns an OAuth 2.0 access token (`access_token`), which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJE1NTExMTIzNzYsInVzZXJfbmFtZSI6InBldGVyIiwiYYV0aG9yaXRpZXMiOlssiUk9MRV9VU0VSI10sImp0aSI6IjRkMmjNjQ4LTQ2MWQtNGV1Y1hZTljlTV1YWUxZjA4ZTJhMiIsImNsawVudF9pZCIE6ImFwcGxpY2F0aw9uaWQiLCJzY29wZSI6NyJmb28iXX0.tr4yUmGLtsH7q9Ge2i7gxyTsOoa0RS0Yoc2uBuAW50VIKZcVsIITWV3bDN0FVHBzimpAPy33tvicFROhBFoVThqKXzzG00SkURN5bnQ4uFLAP0NpZ6BuDjvVmwxNxNrQp21VX141Q4eTvuyZozjUSCxzCI1Lnw5EFFi22J73g1_mRm2j-dEhBp1TvmRaRKLBdk2hzIDVKzu5oj_gODBFm3aLS-IJjYoCimIm2igcesXkipRjtjNcrJSegBbGgyXHVak2gB7I07ryVw1_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97j1T6Rr1XZBlveyjfKs_XIXgu5qizRm9mt5xg", "token_type": "bearer", "refresh_token": "", "expires_in": 5999, "scope": "foo", "jti": "4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

Now try to invoke the Order Processing microservice with the JWT you got from the above curl command. Set the same JWT we got in the HTTP Authorization Bearer header using the following curl command and invoke the Order Processing microservice. Because the JWT is little lengthy, you can use a small trick when using the curl command. First, export the JWT to an environmental variable (TOKEN). Then use that environmental variable in your request to the Order Processing microservice. Once again, here too we use the environment variables, which we defined before for the hostname and the port of the `istio-ingressgateway` service. Since we are using Server Name Indication (SNI) based routing at the Istio gateway, we are using the host name, `orders.ecomm.com` to access the Order Processing microservice. Since there is no DNS mapping to this hostname, we are using the `--resolve` parameter in cURL to define the hostname to IP mapping.

```
\> export TOKEN=jwt_access_token
\> curl -k -H "Authorization: Bearer $TOKEN" --resolve orders.ecomm.com:$INGRESS_HTTPS_PORT:$INGRESS_HOST https://orders.ecomm.com:$INGRESS_HTTPS_PORT/orders/11
```

```
{"customer_id": "101021", "order_id": "11", "payment_method": {"card_type": "VISA", "expiration": "01/22", "name": "John Doe", "billing_address": "201, 1st Street, San Jose, CA"}, "items": [{"code": "101", "qty": 1}, {"code": "103", "qty": 5}], "shipping_address": "201, 1st Street, San Jose, CA"}
```

C.10.4 Debugging Envoy proxy

With the Istio service mesh architecture all the requests coming in and going out of your microservices go through the Envoy proxy. If something goes wrong in your microservices deployment, having access to the debug level logs of the Envoy proxy gives you more visibility and helps troubleshooting. By default, debug level logs are not enabled. You can run the following command with the correct label of the pod (which the Envoy proxy is attached) to enable debug level logs on the Envoy proxy. Here we use `orders` as the value of the `app` label, which you can find in the Deployment definition of the Order Processing microservice. If you want to do the same for the Inventory microservice, then use `app=inventory`.

```
\> kubectl exec $(kubectl get pods -l app=orders -o jsonpath='{.items[0].metadata.name}') -c istio-proxy -- curl -X POST "localhost:15000/logging?filter=debug" -s
```

Now, to view logs, use the following command with name of the Order Processing Deployment. Here the `istio-proxy` is the name of the container, which runs the Envoy proxy.

```
\> kubectl logs orders-deployment-6d6cd77c6-fc8d5 -c istio-proxy --follow
```

In case you have a larger log file, and if it takes time to load, you can use the following command to view the last 100 lines of the log file.

```
\> kubectl logs orders-deployment-6d6cd77c6-fc8d5 -c istio-proxy --tail 100
\> kubectl exec $(kubectl get pods -l app=istio-ingressgateway -n istio-system -o jsonpath='{.items[0].metadata.name}') -n istio-system -c istio-proxy -- curl -X POST "localhost:15000/logging?filter=debug" -s
```

D

OAuth 2.0 and OpenID Connect

OAuth 2.0 is an authorization framework developed by Internet Engineering Task Force (IETF) OAuth working group. It is defined in the RFC 6749 (<http://bit.ly/2RsroHe>). The fundamental focus of OAuth 2.0 is to fix the **access delegation** problem. OpenID Connect is an identity layer built on top of OAuth 2.0, and the OpenID Foundation developed the OpenID Connect specification. In chapter 2 we briefly discussed OAuth 2.0 and how to use OAuth 2.0 to protect a microservice and do service-level authorization with OAuth 2.0 scopes. Then in chapter 3 we discussed how to use Zuul API gateway to do OAuth 2.0 token validation. In chapter 4 we discussed how to login to a single-page application with OpenID Connect and then access the Order Processing microservice, which is protected with OAuth 2.0. In this appendix, we delve deep into OAuth 2.0 and OpenID Connect fundamentals that you need to understand as a microservices developer. If you are interested in understanding OAuth 2.0 and API security in detail, we would recommend you to have a look at the book, Advanced API Security: OAuth 2.0 and Beyond (Apress, 2019) by Prabath Siriwardena (a co-author of this book). OAuth 2 in Action (Manning Publications, 2017) by Justin Richer and Antonio Sanso is also a very good reference on OAuth 2.0.

D.1 The access delegation problem

If you want someone else to access a resource (a microservice, an API, and so on) on your behalf, you need to delegate the corresponding access rights to access the corresponding resource to that person (or thing). For example, if you want a third-party application to read your Facebook status messages, then you need to give that third-party application the corresponding rights to access the Facebook API. There are two models of access delegation as listed below.

- Access delegation via credential sharing
- Access delegation with no credential sharing

If we follow the first model, then you need to share your Facebook credentials with the third-party application so it can use the Facebook API, authenticate with your credentials, and read your Facebook status messages. This is a quite dangerous model. Once you share your credentials with the third-party application, it can do anything – not just read your Facebook status messages. It can read your friends list, view your photos, and chat with your friends via Messenger. This is the model many applications used before OAuth. FlickrAuth, Google AuthSub, Yahoo BBAuth all tried to fix this problem, in their own proprietary way, to do access delegation with no credential sharing. OAuth 1.0, released in 2007, was the first effort to crack this problem in a standard way. OAuth 2.0 followed the direction set by OAuth 1.0 and in October 2012 became the RFC 6749.

D.2 How does OAuth 2.0 fix the access delegation problem?

OAuth 1.0 and OAuth 2.0 both fix the access delegation problem, conceptually in the same. The main difference is, OAuth 2.0 is much extensible than OAuth 1.0. OAuth 1.0 is a concrete protocol, while OAuth 2.0 is an extensible authorization framework. In the rest of the appendix, if we just say OAuth, we mean OAuth 2.0.

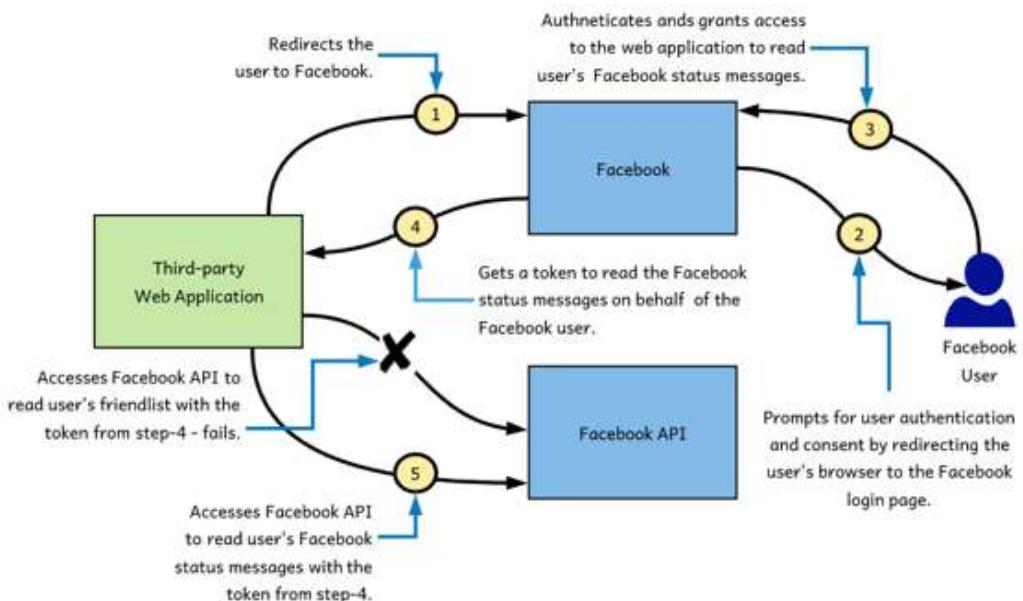


Figure D.1 A third-party application follows the “access delegation with no credential-sharing” model to get a temporary token from Facebook, which is only good enough to read a user’s status messages.

Figure D.1 illustrates a request/response flow where a third-party web application follows the access delegation with the no credential-sharing model to get access to the Facebook API.

With OAuth 2.0, the third-party web application first redirects the user to Facebook (where the user belongs). Facebook authenticates and gets user consent to share a temporary token with the third-party web application, which is only good enough to read user's Facebook status messages for a limited time period. Once the web application gets the token from Facebook, it uses the token along with the API calls to Facebook.

The temporary token Facebook issues has a limited-lifetime and is bound to the Facebook user, the third-party web application, and the purpose. Under OAuth 2.0 terminology, the Facebook user is called the resource owner; Facebook, which issues the token is called the authorization server; the Facebook API is called the resource server; and the 3rd party web application is called the client. The token Facebook issues to the third-party web application is called the access token, and the purpose of the token is called the scope. The flow of events happens during the process of the third-party web application getting the token is called a grant flow, which is defined by a grant type. In the rest of the appendix, we discuss these concepts in detail.

D.3 Actors of an OAuth 2.0 flow

In OAuth 2.0, we mainly talk about four actors, based on the role each plays in an access delegation flow, as listed below (see figure D.2).

- The resource server
- The client
- The end-user (also known as the resource owner)
- The authorization server

In a typical access delegation flow, a client accesses a resource that is hosted on a resource server on behalf of an end-user (or a resource owner), with a token provided by an authorization server. This token grants access rights to the client to access the resource on behalf of the end-user.

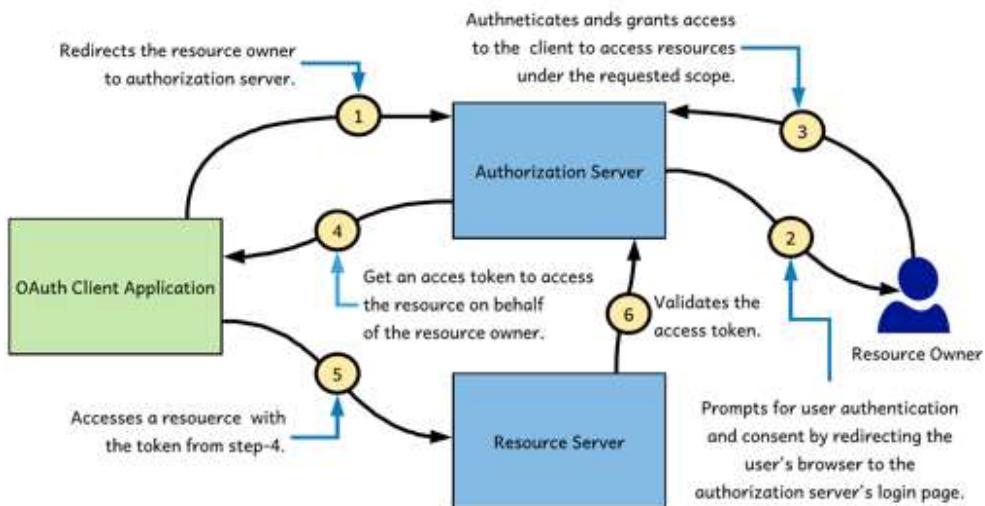


Figure D.2 Actors in an OAuth2.0 flow. In a typical access delegation flow, a client accesses a resource that is hosted on a resource server on behalf of the end-user with a token provided by the authorization server.

D.3.1 The role of the resource server

The resource server hosts the resources and decides who can access, which resources based on which conditions. If we take Flickr, the famous image and video hosting service, all your images and videos that you upload to Flickr are resources. Since Flickr hosts them all, Flickr is the resource server. In a microservices deployment, we can consider a microservice, for example the Order Processing microservice that you developed and tested earlier in the book as a resource server and the orders as the resources. Or in other words, the Order Processing microservice is the entity that is responsible for managing orders. Also you can consider the API gateway that exposes all your microservices to the external client applications as a resource server. As we discussed in chapter 5, the API gateway enforces throttling and access control policies centrally against all the APIs it hosts.

D.3.2 The role of the client application

The client is the consumer of the resources. If we extend the same Flickr example that we discussed before, a web application that wants to access your Flickr photos is a client. It can be any kind of an application: a mobile, web or even a desktop application. In a microservices deployment, the application from which you'd consume the Order Processing microservice is the client application. The client application is the entity in an OAuth flow, which seeks end-user's approval to access a resource on his/her behalf.

D.3.3 Who is the resource owner?

The resource owner (or the end-user) is the one who owns the resources. In the same Flickr example we discussed before, you are the resource owner (or the end-user), who owns your Flickr photos. In a microservices deployment, the person who places orders via the client application (which internally talks to the Order Processing microservice) is the end-user. In some cases, the client application itself can be the end-user, where it simply accesses the microservice, just being itself and no other party involved in.

D.3.4 The role of the authorization server (AS)

In an OAuth 2.0 environment, the authorization server issues tokens (commonly known as access tokens). An OAuth 2.0 token is a key issued by an authorization server to a client application to access a resource (for example a microservice or an API) on behalf of an end-user. The resource server talks to the authorization server to validate the tokens come along with the access requests. The authorization server should know how to authenticate the end-user, as well as to validate the identity of the client application before issuing an access token.

D.4 Grant types

In this section, we talk about OAuth 2.0 grant types and show you how to pick the correct grant type for your applications. Since this book is on microservices, we focus our discussion on microservices, but please keep in mind that OAuth 2.0 is not just about microservices. Different types of applications bearing different characteristics can consume your microservices. How an application gets an access token to access a resource on behalf of a user, depends on these application characteristics. This request/response flow, the client application picks to get an access token from the authorization server is known as a **grant type**, in OAuth 2.0. The standard OAuth 2.0 specification talks about five main grant types. Each grant type outlines the steps for obtaining an access token. The result of executing a particular grant type is an access token that can be used to access resources on your microservices. Following lists down the five main grant types highlighted in the OAuth 2.0 specification.

- Client credentials (suitable for authentication between two systems with no end-user, as we discuss in section D.4.1)
- Resource owner password (suitable for trusted applications, as we discuss in section D.4.2)
- Authorization code (suitable for almost all the applications with an end-user, as we discuss in section D.4.4)
- Implicit (don't use it!)
- Refresh token (used for renewing expired access tokens, as we discuss in section D.4.3)

The OAuth 2.0 framework isn't restricted just for these five grant types. It's an extensible framework that allows you to add grant types as needed. Following lists down two other popular grant types that aren't defined in the core specification, but in related profiles are.

- SAML bearer (suitable for applications having single sign-on using SAML 2.0, defined in RFC 7522)
- JWT bearer (suitable for applications having single sign-on using OpenID Connect, defined in RFC 7523)

D.4.1 Client credentials grant type

Under client credentials grant type, we only have two participants in the grant flow: the client application, and the authorization server. There is no resource owner. Or in other words, the client application itself is the resource owner. Each client carries its own credentials, known as the client ID and the client secret, issued to it by the authorization server. The client ID is the identifier of the client application; the client secret is client's password. The client application should store and use the client secret securely. For example, you should never store a client secret in cleartext, rather encrypt it and store it in a persistent storage (such as a database). As shown in figure D.3, under client credentials grant type, to get an access token the client application has to send its client ID and client secret to the authorization server over HTTPS. The authorization server validates the combination of the ID and secret, and responds back with an access token.

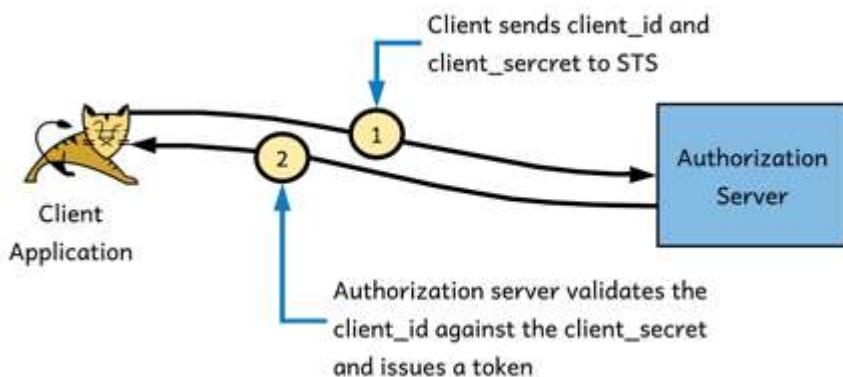


Figure D.3 The client credentials grant type allows an application to obtain an access token, with no end-user – or in other words, the application itself is the end-user.

Here's a sample curl command of a client credentials grant request (this is just a sample, don't try it out as it is).

```
\> curl
-u application_id:application_secret
-H "Content-Type: application/x-www-form-urlencoded"
-d "grant_type=client_credentials" https://localhost:8085/oauth/token
```

The value `application_id` is the client ID, and the value `application_secret` is the client secret of the client application in this case. The `-u` parameter instructs curl to perform a

base64-encoded operation on the string `application_id:application_secret`. The resulting string that's sent as the HTTP Authorization header to the authorization server would be `YXBwbG1jYXRpb25faWQ6YXBwbG1jYXRpb25fc2VjcmV0`. The authorization server validates this request, and issues an access token in the following HTTP response.

```
{"access_token":"de09bec4-a821-40c8-863a-104dddb30204", "token_type":"bearer",
  "expires_in":3599}
```

Even though we used a client secret (`application_secret`) in the above curl command to authenticate the client application to the token endpoint of the authorization server, if required, for stronger authentication the client application can even use mutual Transport Layer Security (mTLS), instead of the client secret. In that case, we need to have a public/private key pair at the client application end, and the authorization server must trust the issuer of the public key or the certificate.

The client credentials grant type is mostly suited for applications to access APIs that do not worry about an end user. Simply, its good for cases where you need not to worry about access delegation or in other words, the client application accesses an API, just by being itself – not on behalf of anyone else. Therefore, the client credentials grant type is mostly used for system-to-system authentication when an application, a periodic task, or any kind of a system directly wants to access your microservice over OAuth 2.0. Let's take a weather microservice for example, which provides weather predictions for next 5 days. If you build a web application to access the weather microservice, then you can simply use the client credentials grant type, because the weather microservice is not interested in knowing who uses your application. It only worries about the application, which accesses it – not the end user.

D.4.2 Resource owner password grant type

The resource owner password grant type is an extension of the client credentials grant type, which adds support for resource owner authentication with the user's username and password. This grant type involves all three parties in the OAuth 2.0 flow, including the resource owner (end user), client application, and authorization server. The resource owner provides the client application his or her username and password. The client application uses this information to make a token request to the authorization server, along with the client ID and client secret embedded within itself. Figure D.4 illustrates the resource owner password grant type.

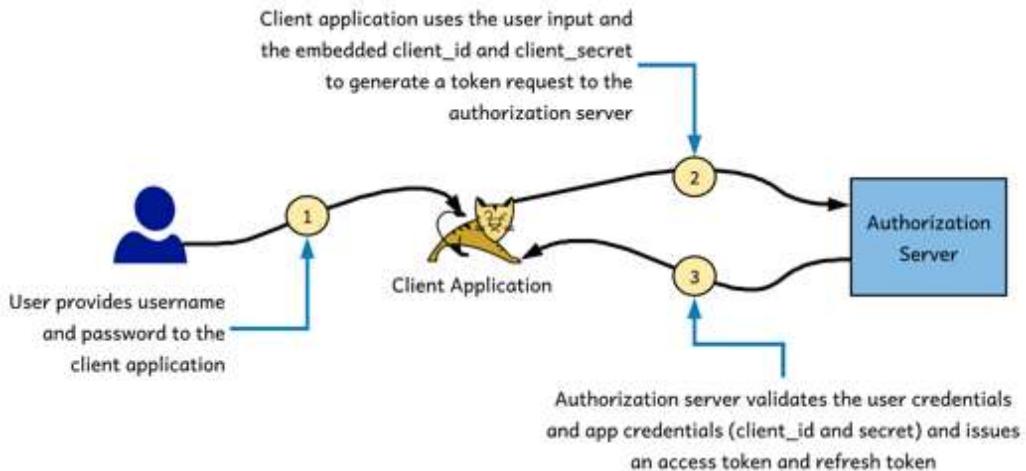


Figure D.4 – The password grant type allows an application to obtain an access token.

Following is a sample curl command of a password grant request made to the authorization server (this is just a sample, don't try it out as it is):

```
\> curl
-u application_id:application_secret
-H "Content-Type: application/x-www-form-urlencoded"
-d "grant_type=password&username=user&password=pass"
'https://localhost:8085/oauth/token'
```

As with the client credentials grant, the application_id and application_secret are sent in base64-encoded form in the HTTP Authorization header. The request body contains the grant type string, the user's username, and the user's password. Note that because you're passing sensitive information in plain-text form in the request header and body, the communication must happen over TLS (HTTPS). Otherwise, any intruder into the network would be able to see the values being passed.

In this case, the authorization server not only validates the client id and secret (application_id and application_secret) to authentication the client application, but also validates the credentials of the user. The issuance of the token happens only if all four fields are valid. As with the client credentials grant type, upon successful authentication, the authorization server responds with a valid access token:

```
{"access_token": "de09bec4-a821-40c8-863a-104dddb30204", "refresh_token": "heasdcu8-as3t-hdf67-vadt5-asdgahr7j3ty3", "token_type": "bearer", "expires_in": 3599}
```

The value of the refresh_token parameter you find in the above response can be used to renew the current access token before it expires. We discuss about refresh token in section D.6. You might have noticed already that we don't get a refresh_token in the client credentials grant type.

Under password grant type, the resource owner (user of the application) needs to provide his or her username and password to the client application. Therefore, this grant type should only be used with client applications that are trusted by the authorization server. This model of access delegation is called, access delegation with credential sharing. It is in fact what OAuth 2.0 wanted to avoid using. Then why it is in the OAuth 2.0 specification? The only reason why the password grant type was introduced in OAuth 2.0 specification was to help the legacy applications using HTTP Basic authentication to migrate to OAuth 2.0. You should try avoiding password grant type where possible.

Like in the client credentials grant type, the password grant type requires the application to store the client secret securely. It's also critically important to deal with the user credentials responsibly. Ideally the client application must not store end user's password locally – just use it to get an access token from the authorization server and forget it. The access token the client application gets at the end of the password grant flow has a limited lifetime. Before this token expires, the client application can get a new token using the `refresh_token` received in the token response from the authorization server. This way, the client application doesn't have to prompt for the user's username and password every time the token on the application expires.

D.4.3 Refresh token grant type

The refresh token grant is used to renew an existing access token. Typically, it's used when the current access token is expired or near expiry, and the application needs a new access token to work with without having to prompt the user of the application to log in again. To use the refresh token grant, the application should receive an access token and a refresh token in the token response. Not every grant type issues a refresh token along with its access token, including the client credentials grant and the implicit grant (discussed later in this chapter). Therefore, the refresh token grant type is a special grant type that can be used only with applications that use other grant types to obtain the access token. Figure D.5 illustrates the refresh token grant flow.

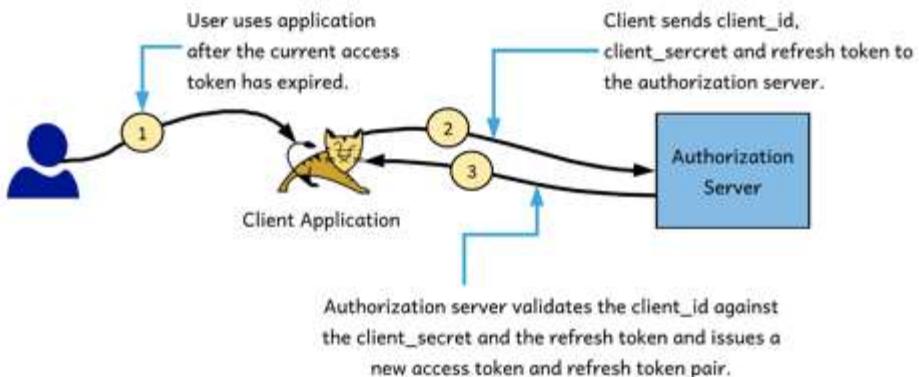


Figure D.5 The refresh token grant type allows a token to be renewed when expired.

The following curl command can be used to renew an access token with the refresh token grant (this is just a sample, don't try it out as it is).

```
\> curl
-u application_id:application_secret
-H "Content-Type: application/x-www-form-urlencoded"
-d "grant_type=refresh_token&refresh_token=heasd...7j3ty3"
'https://localhost:8085/oauth/token'
```

As in the earlier cases, the client ID and client secret (`application_id` and `application_secret`) of the application must be sent in base64-encoded form as the HTTP Authorization header. You also need to send the value of the refresh token in the request payload (body). Therefore, the refresh token grant should only be used with the applications that can store the client secret and refresh token values securely, without any risk of compromise. The refresh token usually has a limited lifetime, but it's generally much longer than the access token lifetime so that an application can renew its token even after significant duration of idleness. When you refresh an access token, in the response, the authorization servers sends the renewed access token, along with another refresh token. This refresh token may be or may be not the same refresh token you got in the very first request from the authorization server. It's up to the authorization server – and not governed by the OAuth 2.0 specification.

D.4.4 Authorization code grant type

The authorization code grant is used in web applications (accessed via a web browser) or native mobile applications that are capable of handling HTTP redirects or even with desktop applications. In the authorization code grant flow, the client application first initiates an authorization code request to the authorization server. This request provides the client ID of the application and a redirect URL to redirect the user when authentication is successful. Figure D.6 illustrates the flow of the authorization code grant.

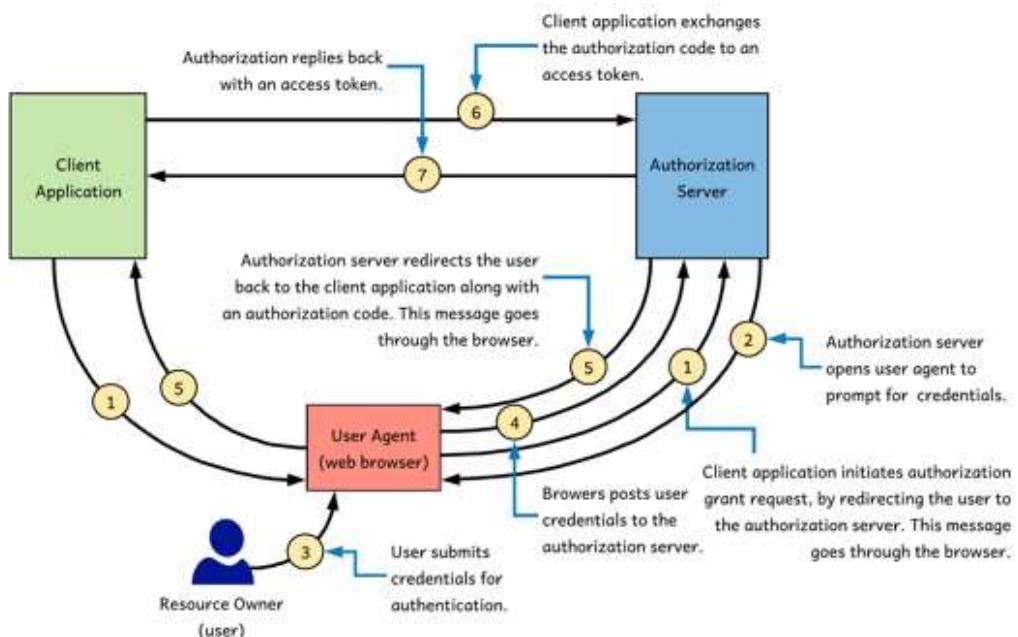


Figure D.6 The authorization code grant type allows a client application to obtain an access token on behalf of an end user (or a resource owner).

As shown in figure D.6, the first step of the application is initiating the authorization code request. The HTTP request to get the authorization code looks like the following (this is just a sample, don't try it out as it is).

```
GET https://localhost:8085/oauth/authorize?
    response_type=code&
    client_id=application_id&
    redirect_uri=https%3A%2F%2Fweb.application.domain%2Flogin
```

As you can see, the above request carries the `client_id` (`application_id`), the `redirect_uri`, and the `response_type` parameters. The `response_type` indicates to the authorization server that an authorization code is expected as the response to this request. This authorization code is provided as a query parameter in an HTTP redirect (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections>) on provided the `redirect_uri`. The `redirect_url` is the location to which the authorization server should redirect the browser (user agent) upon successful authentication. In HTTP, a redirect happens when the server sends a response code between 300 and 310. In this case, the response code would be 302. The response would contain an HTTP header named `Location`, and the value of the `Location` header would bear the URL to which the browser should redirect. A sample `Location` header looks like this:

```
Location: https://web.application.domain/login?code=hus83nn-8ujq6-7snuelq
```

This `redirect_url` should be equal to the `redirect_url` provided when registering the particular client application on the authorization server. The URL (host) in the `Location` response header should be equal to the `redirect_url` query parameter in the HTTP request used to initiate the authorization grant flow. One optional parameter that's not included in the authorization request above is `scope`. When making the authorization request, the application can request the scopes it requires on the token to be issued. We discuss scopes in detail under the section D.5.

Upon receiving this authorization request, the authorization server first validates the client ID and the redirect URL; if these items are valid, it presents the user the login page of the authorization server (assuming that no valid user session is already running on the authorization server). The user needs to enter his username and password on this login page. When the username and password are validated, the authorization server issues the authorization code and provides it to the user agent via an HTTP redirect. The authorization code is part of the redirect URL as shown below.

```
https://web.application.domain/login?code=hus83nn-8ujq6-7snuelq
```

Because the authorization code is provided to the user agent via the redirect URL, it must be passed over HTTPS. Also, since this is a browser redirect, the value of the authorization code is visible to the end-user and also may be logged in server logs. To reduce the risk that this data will be compromised, the authorization code usually has a short lifetime (no more than 30 seconds) and is a one-time-use code. If the code is used more than once, the authorization server revokes all the tokens previously issued against it. Upon receiving the authorization code, the client application issues a token request to the authorization server, requesting an access token in exchange for the authorization code. Following is a curl command of such a request (step 6 in figure D.6).

```
\> curl
-u application1:application1secret
-H "Content-Type: application/x-www-form-urlencoded"
-d "grant_type=authorization_code&
code=hus83nn-8ujq6-7snuelq&
redirect_uri=https%3A%2F%2Fweb.application.domain%2Flogin"
https://localhost:8085/oauth/token
```

Like the other grant types discussed so far, the authorization code grant type too requires the client ID and client secret (optional) to be sent as an HTTP Authorization header in base64-encoded format. It also requires the `grant_type` parameter to be sent as `authorization_code`; the value of the code itself and the `redirect_url` are sent in the payload of the HTTP request to the authorization server's token endpoint. Upon validation of these details, the authorization server issues an access token to the client application in an HTTP response:

```
{"access_token": "de09bec4-a821-40c8-863a-104dddb30204", "refresh_token": "heasdcu8-as3t-hdf67-vadt5-asdgahr7j3ty3", "token_type": "bearer", "expires_in": 3599}
```

Prior (step 5 in figure D.6) to returning an authorization code, the authorization server validates the user by verifying the user's username and password. In step 6, the authorization server validates the client application by verifying the application's client ID and secret. The authorization code grant type does not mandate authenticating the application. So, it is not a must to use the application secret in the request to the token endpoint to exchange the authorization code to an access token. This is the recommended approach when you use the authorization code grant type with single page applications (SPAs), which we discussed in chapter 4.

As you've seen, the authorization code grant involves the user, client application, and authorization server. Unlike the password grant, this grant type doesn't require the user to provide his or her credentials to the client application. The user provided his credentials only on the login page of the authorization server. This way, you prevent the client application from knowing the user's login credentials. Therefore, this grant type is suitable for web, mobile and desktop applications that you don't fully trust to provide user credentials to.

A client application that uses this grant type needs to have some prerequisites to use this protocol securely. Because the application needs to know and deal with sensitive information such as the client secret, refresh token, and authorization code, it needs to be able to store and use these values with caution. It needs to have mechanisms for encrypting the client secret and refresh token at storage and to use HTTPS for secure communication with the authorization server, for example. The communication between the client application and the authorization server needs to happen over TLS so that network intruders don't see the information being exchanged.

D.4.5 Implicit grant type

The implicit grant type is similar to the authorization code grant type. But doesn't involve the intermediary step of getting an authorization code before getting the access token. Instead, the authorization server issues the access token directly in response to the implicit grant request. Figure D.7 illustrates the implicit grant flow.

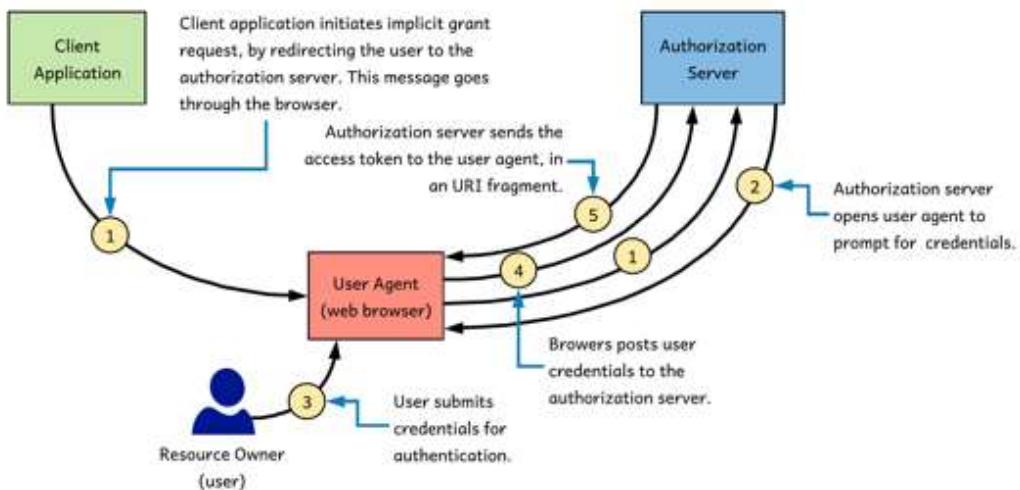


Figure D.7 The implicit grant type allows a client application to obtain an access token.

With implicit grant type, when the user attempts to log in to the application, the client application initiates the login flow by creating an implicit grant request. This request should contain the client ID and the redirect URL. The redirect URL, as in the authorization code grant type, is used by the authorization server to redirect the user agent back to the client application when authentication is successful. Following is a sample implicit grant request (this is just a sample, don't try it out as it is).

```
GET https://localhost:8085/oauth/authorize?
  response_type=token&
  client_id=application_id&
  redirect_uri=https%3A%2F%2Fweb.application.domain%2Flogin
```

As you can see in the above HTTPS request, the difference between the authorization code grant's authorize request and the implicit grant's implicit request is the fact that the `response_type` parameter in this case is `token`, which indicates to the authorization server that you're interested in getting an access token as the response to the implicit request. As in the authorization code grant here too, `scope` is an optional parameter that the user agent may provide to ask the authorization server to issue a token with the required scopes.

When the authorization server receives this request, it validates the client ID and the redirect URL, and if those are valid, it presents the user the login page of the authorization server (assuming that no active user session is running on the browser against the authorization server). When the user enters his or her credentials, the authorization server validates them and presents to the user the consent page to acknowledge that the application is capable of performing the actions denoted by the `scope` parameter (only if `scope` parameter is provided in the request). Note that the user provides credentials on the login page

of the authorization server, so only the authorization server gets to know the user's credentials. When the user has consented to the required scopes, the authorization server issues an access token and provides it to the user agent on the redirect URL itself as a URI fragment. Following is an example of such a redirect:

```
https://web.application.domain/login#access_token=jauej28s1ah2&
expires_in=3599
```

When the user agent (web browser) receives this redirect, it makes an HTTPS request to the `web.application.domain/login` url. Because the `access_token` field is provided as a URI fragment (denoted by the `#` character in the URL), however, that particular value doesn't get submitted to the server on `web.application.domain`. Only the authorization server (which issued the token) and the user agent (web browser) get to know the value of the access token. The implicit grant doesn't provide a refresh token to the user agent. As we discussed earlier in this chapter, because the value of the access token is passed in the URL, it will be in the browser history and also possibly logged into server logs.

The implicit grant type doesn't require your client application to maintain any sensitive information, such as a client secret or refresh token. This fact makes it a good candidate to be used in single-page applications (SPAs) in which rendering of the content happens on web browsers through JavaScript. These types of applications execute mostly on the client side (browser). Therefore, they're incapable of handling sensitive information such as client secrets. But still, the security concerns in using implicit grant type is much higher than its benefits, and it is no longer recommended, even for SPAs. As discussed before in the previous section, even for SPAs, the recommendation is to use the authorization code grant type, with no client secret.

D.5 Scopes bind capabilities to an OAuth 2.0 access token

Each access token an authorization server issues, is associated with one or more scopes. A scope defines the purpose of a token. A token can have more than one purpose, hence it can be associated with multiple scopes. In other words, a scope defines what the client application can do at the resource server with the corresponding token. When a client application requests a token from the authorization server, along with the token request, it also specifies the scopes, it expects from the token (see figure D.8). That necessarily does not mean the authorization server has to respect that request and issue the token with all requested scopes. Authorization server can decide on its own, also with the resource owner's consent, which scopes to associate with the access token. In the token response, it will send back to the client application, the scopes associated with the token along with the token.

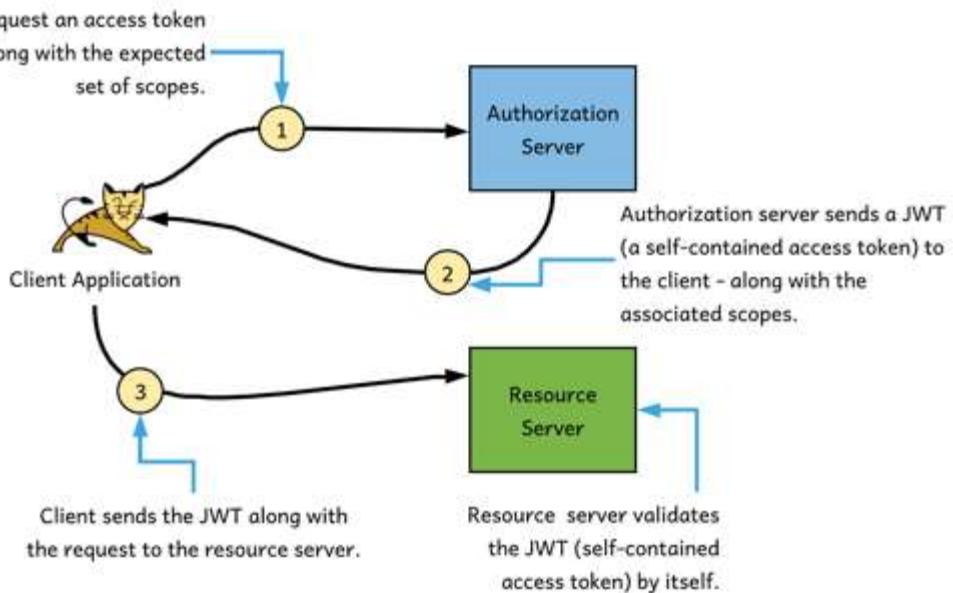


Figure D.8 Client application requests an access token along with the expected set of scopes. When the access token is a self-contained JWT, the resource server validates the token by itself, without talking to the authorization server.

D.6 Self-contained access tokens

An access token can be either a reference token or a self-contained token. A reference token is just a string – and only the issuer of the token knows how to validate it. When the resource server gets a reference token, it has to talk to the authorization server all the time to validate the token. In contrast, if it is a self-contained token, the resource server can validate the token itself – no need to talk to the authorization (see figure D.8). A self-contained token is a signed JWT or a JWS (see appendix H). The JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens profile (which is in its first draft at the time of writing), developed under IETF OAuth working group defines the structure for a self-contained access token.

D.7 What is OpenID Connect?

OpenID Connect is built on top of OAuth 2.0 as an additional identity layer. It uses the concept of an *ID token*. An *ID token* is a JSON Web Token (JWT) that contains authenticated user information, including user claims and other relevant attributes. When an authorization server issues an ID token, it signs the contents of the JWT (a signed JWT is called JWS or JSON Web Signature), using its private key. Before any application accepts an ID token as valid, it should verify its contents by validating the signature of the JWT.

NOTE An *ID token* is consumed by an application to get information such as a user's username, email address, phone number, and so on. An *access token* is a credential used by an application to access a secured API on behalf of an end-user or just by being itself. OAuth 2.0 only provides an access token, while OpenID Connect provides both an access token and an ID token.

Following is an example of a decoded ID token (payload only) that includes the standard claims as defined by the OpenID Connect (OIDC) specification (https://openid.net/specs/openid-connect-core-1_0.html#IDtoken):

```
{
  "iss": "http://server.example.com",
  "sub": "janedoe@example.xom",
  "aud": "8ajduw82swiw",
  "nonce": "82jd27djuw72jduw92ksury",
  "exp": 1311281970,
  "iat": 1311280970,
  "auth_time": 1539339705,
  "acr": "urn:mace:incommon:iap:silver",
  "amr": "password",
  "azp": "8ajduw82swiw"
}
```

Details on these attributes are in the OIDC specification. Following are a few important ones:

- **iss**—The identifier of the issuer of the ID token (usually, an identifier to represent the authorization server that issued the ID token)
- **sub**—The subject of the token for which the token was issued (usually, the user who authenticated at the authorization server)
- **aud**—The audience of the token; a collection of identifiers of entities that are supposed to use the token for a particular purpose. It must contain the OAuth2.0 `client_id` of the relying party (the SPA), and zero or more other identifiers (an array). If a particular SPA is using an ID token, it should validate whether it's one of the intended audiences of the ID token. In other words, the SPA's `client_id` should be one of the values in the `aud` claim.
- **iat**—The time at which the ID token was issued.
- **exp**—The time at which the ID token expires. An application must use an ID token only if its `exp` claim is later than the current timestamp.

An ID token usually is obtained as part of the access token response. OAuth2.0 providers support various grant types for obtaining access tokens, as we discussed in the section D.5. An ID token usually is sent in the response to a request for an access token using a grant type. You need to specify `openid` as a scope in the token request to inform the authorization server that you require an ID token in the response. The following is an example of how to request an ID token in an authorization request when using the `authorization_code` grant type:

```
GET https://localhost:8085/oauth/authorize?
    response_type=code&
    scope=openid&
    client_id=application1&
```

```
redirect_uri=https%3A%2F%2Fweb.application.domain%2Flogin
```

The ID token is sent in the response to the token request in following form:

```
{  
  "access_token":  
    "sdfj82j7sjej27djwtterh720fnwqudkdnw72itjswnrlvod92hvkwfyfp",  
  "expires_in": 3600,  
  "token_type": "Bearer",  
  "id_token": "sdu283ngk23rmas...."  
}
```

`id_token` is a JWT, which is built with three base64-url-encoded strings, each separated by a period. We omitted the full string in the example for readability. In chapter 3, you see how to use OpenID Connect in practice, with a Single-page Application (SPA).

E

Single-page application architecture

In chapter 4, we discussed how to create a single-page application (SPA) with Angular and then talked about the Order Processing microservice from the SPA. SPA is already a popular architectural pattern for building applications against a set of APIs. In fact, the rise of API adoption had a great influence in moving developers to build SPAs. In this appendix, we discuss the basic principles behind the SPA architecture.

E.1 What is single-page application (SPA) architecture?

A single-page application (SPA) is an architectural pattern used to develop frontend, user-facing web applications. In a traditional multiple-page application (MPA) architecture, when a web browser makes a request to the web server, the web server first loads the data (content) required for the requested web page (by reading a database, talking to other external services, and so on), generates the HTML content, and provides it to the web browser for rendering.

Notice the word *HTML* in the preceding sentence. In this case, the server is responsible for generating the HTML for the browser to render, which is why these types of applications are known as multi-page applications. As illustrated in figure E.1, when the web browser makes a request for a particular page, the web server requests data from a data source and generates the HTML content using that data. This HTML content is then sent back to the web browser.

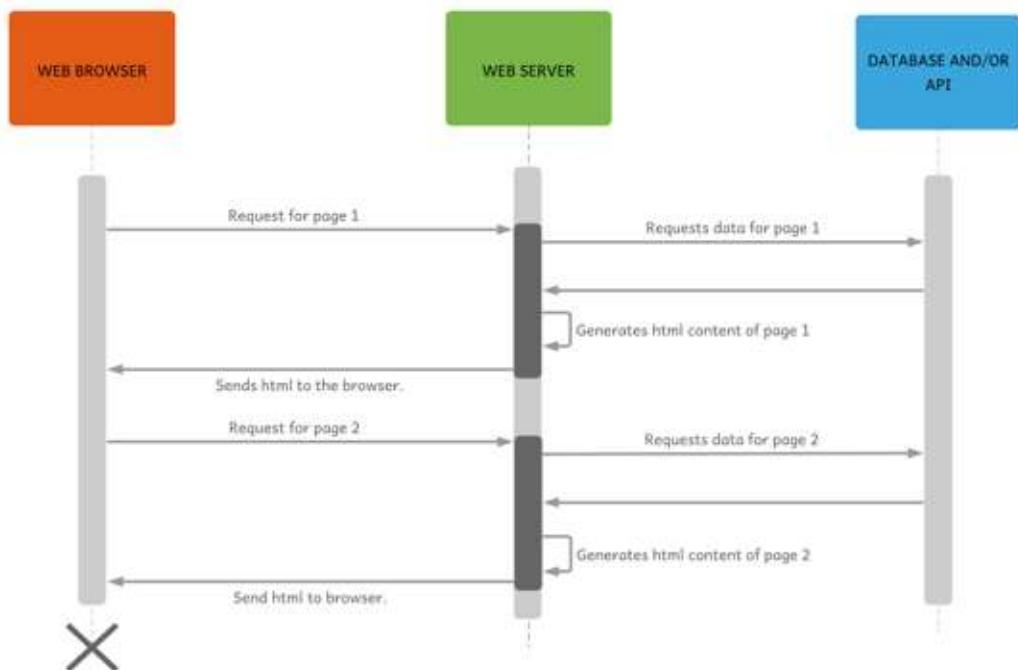


Figure E.1 An MPA loads content to the browser with multiple page reloads.

A SPA, on the other hand, loads the initial HTML, cascading style sheets (CSS), and JavaScript to the browser when loading the application for the first time. On requests to fetch further data, it directly downloads the actual data (JavaScript Object Notation [JSON] or whatever the data format is) from the web server. The generation of dynamic HTML content happens on the browser itself through the JavaScript that's already loaded (and cached). Figure E.2 illustrates the flow of actions for a SPA.

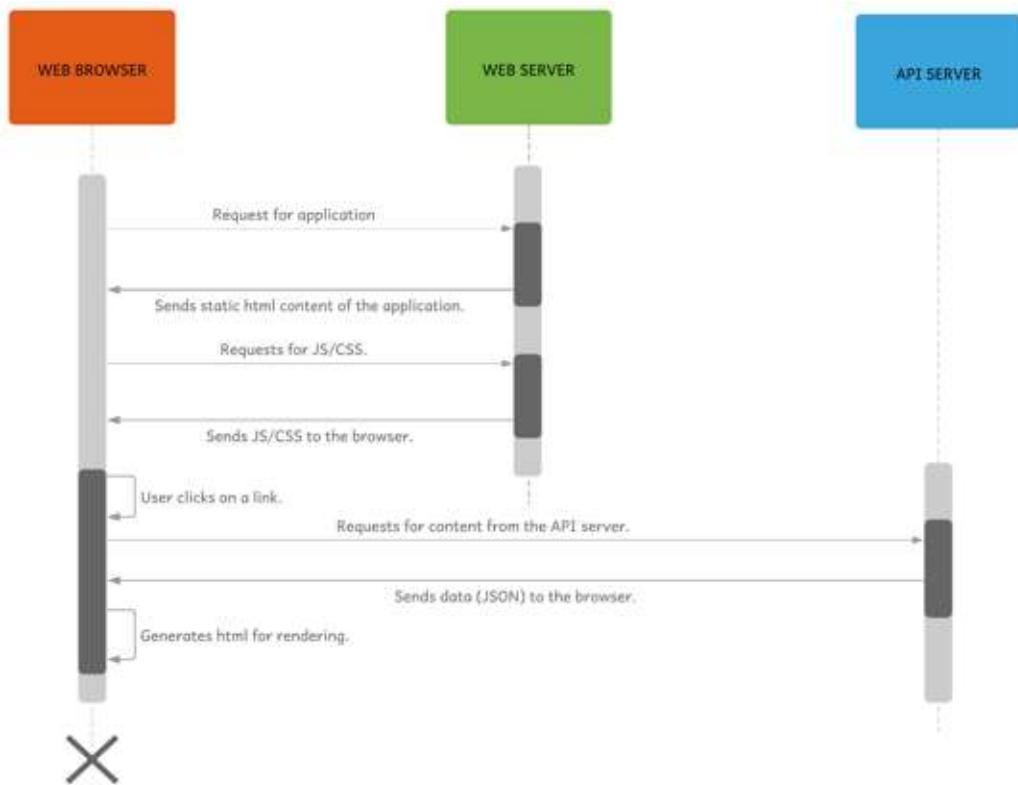


Figure E.2 Any web application that follows this architectural pattern is known as a SPA. A SPA loads content to the browser with no page reloads.

Most of the modern websites you use today follow the pattern shown in figure E.2. If you've used Gmail, Google Maps, Facebook, Twitter, Airbnb, Netflix, PayPal, and the like, you've used a SPA. All these websites use this design to load content into your web browser.

A typical difference between an MPA and a SPA is that in an MPA, each request for new content reloads the web page. But in a SPA, after the application has been loaded into the browser, no page reloads happen. All new content is rendered on the browser itself without any requests to the web server. The SPA talks to different endpoints (APIs) to retrieve new content, but the rendering of that content happens in the browser.

The reason these are called *single-page applications* is that most of them have only one HTML file (a single page) for the entire application. The content of the website is rendered by dynamically changing the HTML of the file upon user actions. You'll see how this works in examples later in the appendix.

E.2 Benefits of a SPA over an MPA

A SPA has several benefits compared to an MPA, some of which are specifically useful for microservice architectures:

- Beyond the initial loading of the application, page rendering is faster because the generation of HTML happens on the client side (browser) and the amount of data being downloaded is reduced (no HTML; mostly JSON content). The HTML, CSS, and JavaScript are loaded once throughout the lifespan of the application.
- The load on the web server is reduced because the server has been relieved of the responsibility to generate HTML content. This thereby reduces the need for the web application to scale, saving a lot of problems related to handling sessions.
- Because the application design is simple (HTML, CSS, and JavaScript only), the application can be hosted in any type of environment that can be accessed over HTTP and doesn't require advanced web server capabilities.
- The application becomes more flexible because it can easily be changed to talk to any number of microservices (via APIs) for fetching data and rendering as appropriate.
- Because SPAs retrieve data mostly from standard HTTP-based REST APIs, they can cache data effectively and use that data offline. A well implemented REST API supports HTTP ETags¹ and similar cache validations, making it easy for browsers to store, validate, and use client-side caches effectively.

E.3 Drawbacks of a SPA compared with an MPA

SPAs don't come for free, however. They have drawbacks that you need to think about carefully before committing to implementing them. Luckily, engineers have found ways to overcome the limitations in SPA architectures.

- The rendering of content happens through JavaScript. If the pages contain heavy or unresponsive JavaScript, these can affect the browser process of the application user.
- Because the application relies on JavaScript, it becomes more prone to cross-site scripting (XSS) attacks; therefore, developers have to be extra cautious.
- SPAs won't work on browsers that have JavaScript disabled. There are workarounds for these cases, but these don't help you reap the benefits of a SPA.
- The initial loading of the application into the browser can be slow because it involves loading all the HTML, CSS, and JavaScript. There are ways to improve the loading time by using workarounds, however.
- The application would find it hard to deal with sensitive information such as user credentials or tokens because it works primarily on the client side (browser).

¹An HTTP ETag (entity tag) is one of several mechanisms HTTP provides for web cache validations.

F

Observability in a microservices deployment

In chapter 5, we discussed in detail how to monitor a microservices deployment with Prometheus and Grafana. The modern term for monitoring and analytics is known as *observability*. In this appendix, we discuss the importance of observability in a microservices deployment, and why it's critical to do so, compared to monolithic applications.

F.1 The need for observability

Compared to a traditional monolithic application, microservice-backed applications are heavily distributed. In a traditional monolithic application, when function `foo` calls function `bar`, the chances of the function invocation failing due to external factors are rare. This is because in a monolithic application both functions reside on the same process. If there's a failure in the process, the application will fail as a whole, reducing the chances of partial failures. Figure F.1 illustrates how a traditional retail application is composed of many functions within the same process.

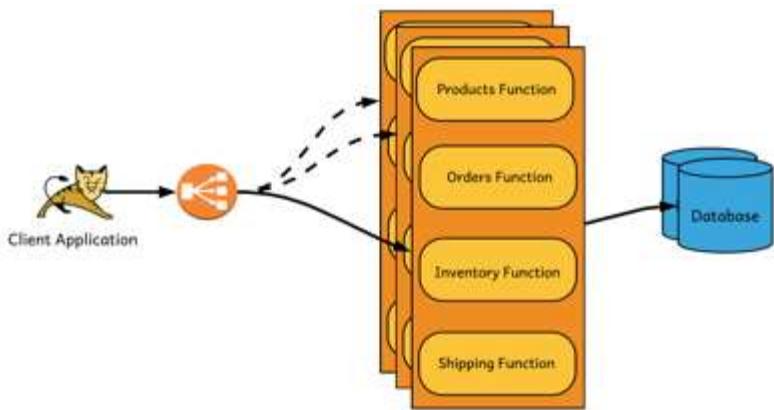


Figure F.1 A scalable monolithic application. All functions of the application are within the same process. A failure of the application results in a failure of all functions.

If we break down the monolith application in figure F.1 into a microservice-driven architecture, we'll probably end up with an application architecture that looks like figure F.2.

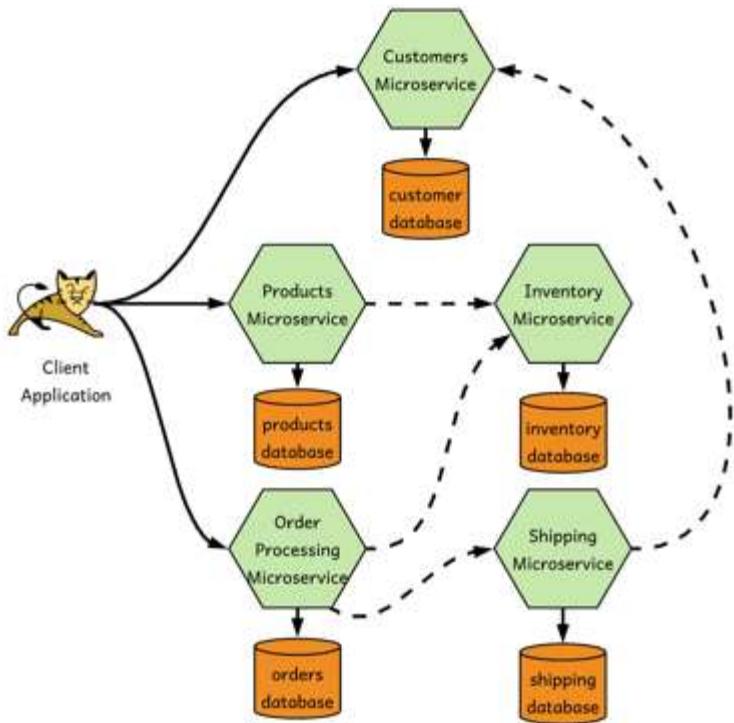


Figure F.2 Microservice-based architecture diagram of the retail store illustrates how individual functions are broken down into independent microservices.

As you can observe from the diagram in figure F.2, a request made from a client can go through multiple hops of microservices in a typical microservices deployment. This calls for resiliency, robustness, and recovery factors to be built into our microservices to minimize failures as much as possible. Let's look at a real example and see why it's important to have stringent monitoring in place.

When a client makes a request to query the available products through the Products microservice, the Products microservice makes a query to the Inventory microservice to get the list of available product stock. At this point, the Inventory microservice can fail due to various reasons such as:

1. The Inventory microservice is running under high resource utilization and, therefore, is slow to respond. This can cause a timeout on the connection between the Products microservice and the Inventory microservice.
2. The Inventory microservice is currently unavailable. The process has either crashed or has been stopped by someone.

3. The Inventory microservice seems pretty healthy in terms of resource utilization; however, it takes a long time to respond due to a few slow-running database queries.
4. Some data inconsistency on the inventory database is causing a failure on the code of the Inventory microservice, resulting in the code being unable to execute successfully, thus generating an error/exception.

In all four types of failures, the products microservice needs to fall back to an alternative, such as presenting the list of products without providing the details on stock availability. As you can see, these types of partial failures of our microservices requires immediate attention to be rectified and fixed. And that's what makes observability a critical factor in our microservices.

F.2 The four pillars of observability

The four main pillars of observability are metrics, tracing, logging, and visualization. Each of these factors are important to monitor our microservices effectively. Let's take a look at why we need to pay attention to each of them.

F.2.1 The importance of metrics in observability

Metrics are a set of data values that are recorded over a period of time. These are mostly numeric values that are constantly recorded by min, max, average, and percentile. Metrics are usually used to measure the efficiency of the software process. These can be things like memory usage of a particular process, CPU usage of a process, load average, and so forth.

If we take a look at the scenario we discussed in the previous section (F.1), metrics come in handy when troubleshooting and taking precautionary actions to minimize the impact of failures described in the first and second points of that section. When a particular microservice is running under heavy resource utilization, monitoring the metrics of the relevant microservice would help to trigger alerts that would enable our DevOps personal to take relevant actions. Systems such as Kubernetes (see appendix B) would monitor these types of metrics to perform auto healing and auto scaling activities so that these failures have minimal business impact.

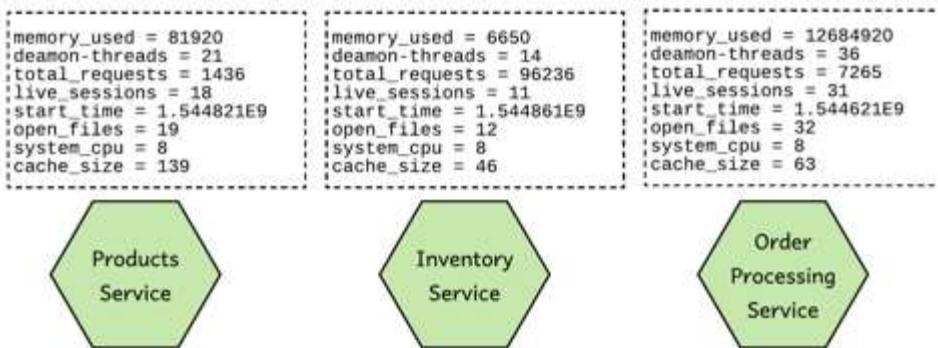


Figure F.3 Recording microservice metrics based on various attributes such as memory, sessions, and so forth

The downside of metrics is that metrics are only useful to monitor a given microservice in isolation, based on a limited set of attributes. There are no technical barriers to adding any number of attributes to your metrics. However, adding a lot of metrics requires a lot of storage, making the microservice harder to manage. Metrics also don't give you a view of the problems that happen due to a request spanning across multiple microservices and other third-party systems. This is when distributed tracing comes to the rescue.

F.2.2 The importance of tracing in observability

A trace is a sequence of related distributed events. A single trace will have a uniquely identifiable identifier (UID), which spans across all the parties involved in the trace. A trace is like a collection of logs that spans across several components of a system. Each log record having a UID that makes it possible to correlate the data of a single request (event), which spans across the various components of the system. Each record in a trace can contain information relevant to tracing and troubleshooting a request, such as entry point timestamps, latency information, or any other information that might be useful to identify the source of a request or to troubleshoot a request flow.

If we take a look at the third point we discussed in section F.1, you'll see that metrics don't help us a lot in that case because the vitals of the systems remain intact. This is a scenario where the Inventory microservice as a whole remains healthy, but a particular function within it that accesses the database takes a longer time to complete, causing the Products microservice to fail partially. If we instrument the code of the Inventory microservice to emit a record with the latency details of the database query, this would help us to identify the particular section in the request flow that causes the problem. Let's take a deeper look at this in figure F.4.

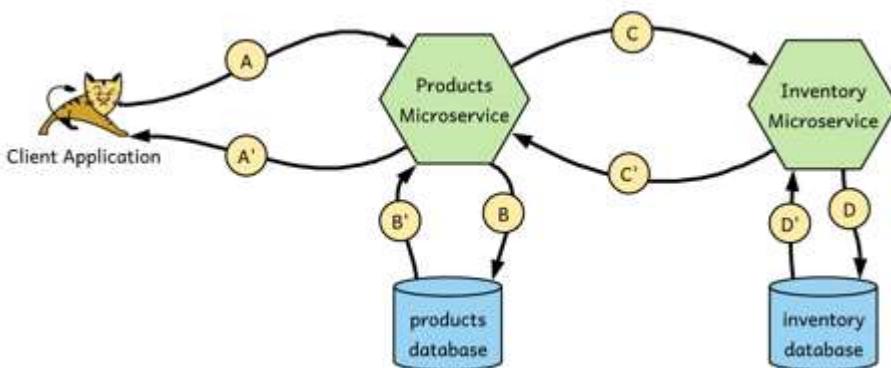


Figure F.4 The points in the request flow at which the spans are started and stopped

As you can see in figure F.4, when a client application makes a request to the Products microservice, it initiates a trace with a UID. The following is a rough sequence of events that happen to serve this request:

1. The client application initiates a request to the Products microservice. Because there's no trace prior to this point, the Products microservice begins the trace by creating the first span (A).¹
2. The Products microservice executes a function to retrieve the list of products from the Products database. The microservice creates a new span for this (B to B') and adds it to the trace started in the previous step. Because this particular span ends at this point itself, the span details are emitted from the microservice here itself.
3. The Products microservice gets the IDs of the relevant products and passes them to the inventory microservice for retrieving the inventory details of each product item. At this point, the microservice creates a new span (C) and adds it to the same trace.
4. The Inventory microservice queries the Inventory database for the inventory details of each product and responds back to the Products microservice. The Inventory microservice creates a new span (D to D') and adds it to the trace started by the Products microservice. Once the function completes, the span details are emitted.
5. The Products microservice creates an aggregated response, which contains the product and inventory details of each product to be sent back to the client application. At this point, the span (C) is completed and the span details are emitted.
6. The Products microservice responds back to the client application with the information. Before sending the response to the client application, the microservice finishes span (A), and the span details are emitted.

Each hop along this flow is represented as a span. When the execution flow reaches a particular point in the instrumentation, a span record is emitted, which contains details about the execution. Each such span would belong to the same trace bearing the UID that was generated by the Products microservice at the point of initiating the processing of the request. If we assume the complete request takes about 1 second to complete, we can think that the database query in the Inventory microservice consumes about 700 milliseconds of that. If that's the case, the spans would look like those in figure F.5.

¹ A span represents an individual unit of work performed in a distributed system.

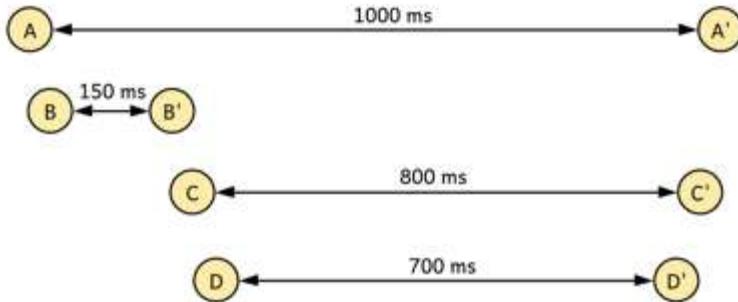


Figure F.5 The time taken in each span. By analyzing these spans, it becomes evident that the database operation that happens in the Inventory microservice consumes the largest chunk of request time.

While the above is the general pattern of tracing distributed events, we would need all our microservices to conform to a single pattern when emitting their traces. This would make it easier and consistent for querying them. As such, it would be perfect to have a global standard for all microservices worldwide when emitting their traces. OpenTracing (<https://opentracing.io/>) is such a vendor-neutral specification for distributed tracing requirements. It provides different instrumentation libraries for various programming languages.

NOTE Jaeger (<https://www.jaegertracing.io/>) and Zipkin (<https://zipkin.io/>) are two of the most popular open source distributed tracing solutions that conform to the OpenTracing specification.

One major challenge with tracing is that it's hard to retrofit it into an existing system. You need to add instrumentation code to a lot of places all across your microservices. And on top of that, it may not even be possible to instrument some of the services involved in the flow, due to those being out of your control. Failure to instrument all components involved in a given flow may not give you the optimal results you're looking for. And in some cases where you're unable to trace an entire flow end-to-end, you may not be able to make use of whatever traces you have in hand as well.

Service meshes (see appendix C) can be a savior in some of these situations where you're unable to instrument your microservices for tracing (or something else). Service meshes attach a sidecar² (figure F.6) to your microservice, which adds tracing capacities (and other capabilities) to it. This way you can strap-on tracing to any microservice without having to modify the microservice at all.

² A sidecar proxy is an architecture pattern that abstracts certain features such as security, traceability away from the main process (microservice).

NOTE Istio and Linkerd are two popular service mesh solutions that can help with your microservices architecture, not just for tracing purposes, but for many other things.

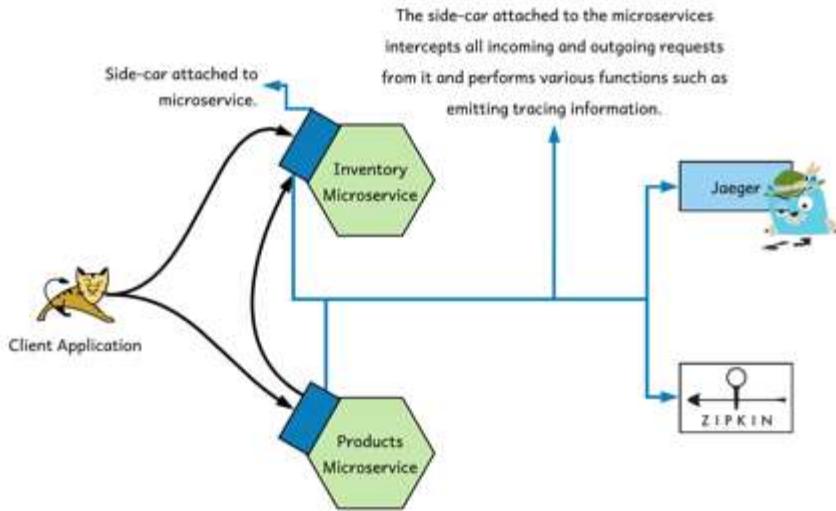


Figure F.6 How sidecars assist microservices to emit tracing-related information without requiring any changes to the microservice themselves

THE IMPORTANCE OF LOGGING IN OBSERVABILITY

A *log* is a timestamped record of a particular event. It could be an acknowledgement of a request notifying the starting of a particular process, the recording an error/exception, and so on. Most readers would, by experience, know the importance of a log record.

Metrics and traces are both used for observing statistical information about your microservices, either in isolation or spanning across several microservices. There are still many instances, however, in which this statistical information isn't useful for identifying the root cause of an issue, as in the failure scenario we discussed under the fourth point in section F.1. In this case, a particular inconsistency in the database was causing a problem in our microservice. This is a type of failure that we may not be able to capture or troubleshoot using metrics or tracing only. From a statistical point of view the microservice would have executed the function without delays or would have added resource usage. However, the code would have failed with exceptions being printed on the logs. This is why it becomes important to monitor your logs using log collection and analysis tools.

Technologies such as Fluentd can be used for aggregation of logs from all of your microservices. It plugs into systems such as ElasticSearch, Splunk, and so on and can be used for the analysis and querying of log records. Fluentd can also be configured to trigger alerts based on connectors to send emails, SMS notifications, and instant messages.

THE IMPORTANCE VISUALIZATION IN OBSERVABILITY

Another key important factor in observability is being able to visualize the data and streams you collect from your microservices. Visualization is, of course, for humans only. We may have automated systems in place that are capable of acting based on failures or risks of failures. But in reality, most organizations still require some levels of human intervention to fix things when they go wrong. Therefore, having dashboards that constantly display the state of your systems can help a great deal. Kibana and Grafana are two popular technologies for visualizing statistics related to monitoring your systems. We discuss using Grafana in detail to monitor microservices in chapter 5. At the time of this writing, Grafana is the most popular open source data visualization tool available.



Open Policy Agent

In a typical microservices deployment, we can enforce access control policies in any of the following two locations or both:

- The edge of the deployment – typically with an API gateway (which we discussed in chapter 5)
- The edge of the service – typically with a service mesh or with a set of embedded libraries (which we discussed in chapter 7 and chapter 12)

Authorization at the service level gives each service more control to enforce access control policies in the way it wants. Typically you apply coarse-grained access control policies at the API gateway at the edge- and more fine-grained access control policies at the service level. Also, it is a common practice to do data level entitlements at the service level. For example, at the edge of the deployment we can check whether a given user is eligible to do an HTTP GET to the Order Processing microservice. But data entitlement checks, such as, only an order admin can view orders having a transaction amount greater than \$10,000, are enforced at the service level.

In this appendix we discuss key components of an access control system, access control patterns, and how to define and enforce access control policies using Open Policy Agent (OPA). OPA (<https://www.openpolicyagent.org/>) is an open source, lightweight general-purpose policy engine that has no dependency on microservices. You can use OPA to define fine-grained access control policies and enforce those policies at different locations in a microservices deployment. We discussed OPA briefly in chapter 5. In this appendix, we will delve deep into the details. We also assume that you have already gone through chapters 5, 7, 10, 11, and 12, and have a good understanding on containers, Kubernetes, Istio and JWT.

G.1 Key components in an access control system

In a typical access control system, we find five key components (figure G.1): policy administration point (PAP), policy enforcement point (PEP), policy decision point (PDP), policy information point (PIP), and policy store. The policy administration point is the component, which lets policy administrators / developers define access control policies. Most of the time PAP implementations come with their own user interface or expose the functionality via an API. Some access control systems do not have a specific PAP, rather it reads policies directly from the file system – so you have to use some 3rd party tools to author policies. Once the policies are developed via a PAP, PAP writes the policies to a policy store. The policy store can be a database, a file system or even a service, which is exposed via HTTP.

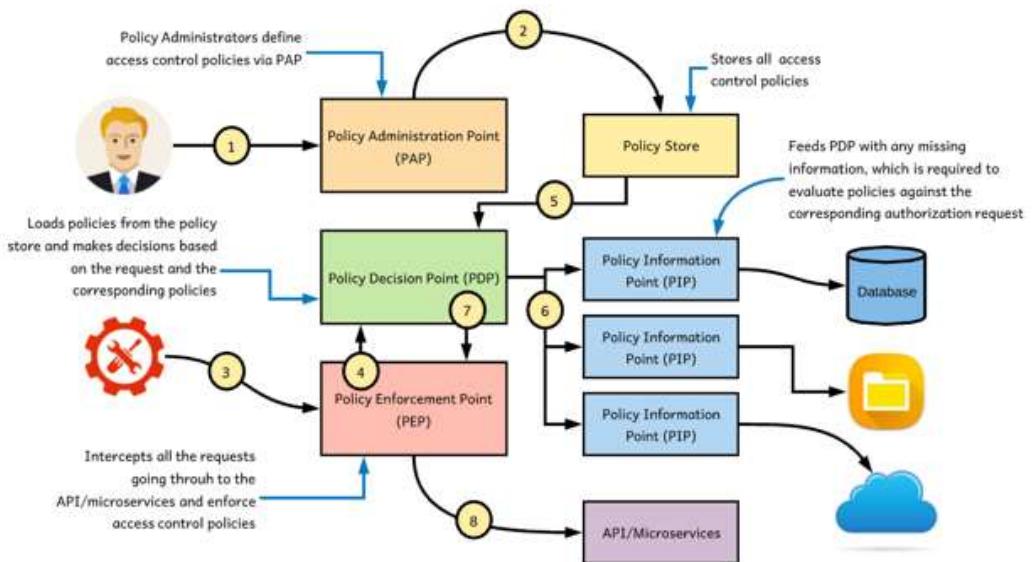


Figure G-1: Components of a typical access control system. PAP defines access control policies and then stored in the policy store. At runtime, the PEP intercepts all the requests, builds an authorization request and talks to the PDP. The PDP loads the policies from the policy store and any other missing information from PIP, evaluates the policies and passes back the decision to the PEP.

The policy enforcement point (PEP) sits between the service/API to be protected and the client application. At runtime, the PEP intercepts all the communications between the client application and the service. As we discussed in chapter 3, the PEP can be an API gateway – or as we discussed in chapter 7 and chapter 8, the PEP can be some kind of an interceptor embedded into your application itself – or as we discussed in chapter 12 in a service mesh deployment, it can be a proxy, which intercepts all the requests coming to your microservice. However when the PEP intercepts a request, it extracts out certain parameters from the request such as the user, resource, action and so on, and creates an authorization request.

Then it talks to the policy decision point (PDP) to check whether the request is authorized or not. Only if it is authorized, will dispatch the request to the corresponding service or the API, otherwise it will return an error to the client application. Before the request hits the PEP, we assume it is properly authenticated.

When the PEP talks to the PDP to check authorization, the PDP loads all the corresponding policies from the policy store. And while evaluating an authorization request against the applicable policies, if there are any required but missing information, the PDP will talk to a policy information point (PIP). For example, lets say we have an access control policy, which says you can only buy a beer if your age is greater than 21, but the authorization request only carries, your name as the subject, buy as the action, and beer as the resource. The age is the missing information here and the PDP will talk to a PIP to find the corresponding subject's age. We can connect multiple PIPs to a PDP – and each PIP can connect to different data sources.

G.2 What is Open Policy Agent (OPA)?

As we discussed in the intro section of the appendix, OPA is an open source, lightweight general-purpose policy engine that has no dependency on microservices. You can use OPA to define fine-grained access control policies and enforce those policies at different places in a microservices deployment. To define access control policies, OPA introduces a new programming language called, Rego. You can find more details about Rego from here: <https://www.openpolicyagent.org/docs/latest/policy-language/>.

OPA started as an open source project in 2016 with a goal to unify policy enforcement across multiple heterogeneous technology stacks. Netflix is one of the early adopters of OPA, who uses OPA to enforce access control policies in its microservices deployment. Apart from Netflix, Cloudflare, Pinterest, Intuit, Capital One and many more use OPA. At the time of this writing OPA is an incubating project under Cloud Native Computing Foundation (CNCF).

G.3 Open Policy Agent (OPA) high-level architecture

In this section we discuss how OPA's high-level architecture fits into the discussion we had in section G.1. As you can see in figure G.2, the OPA engine can run on its own as a stand-alone deployment or as an embedded library along with an application.

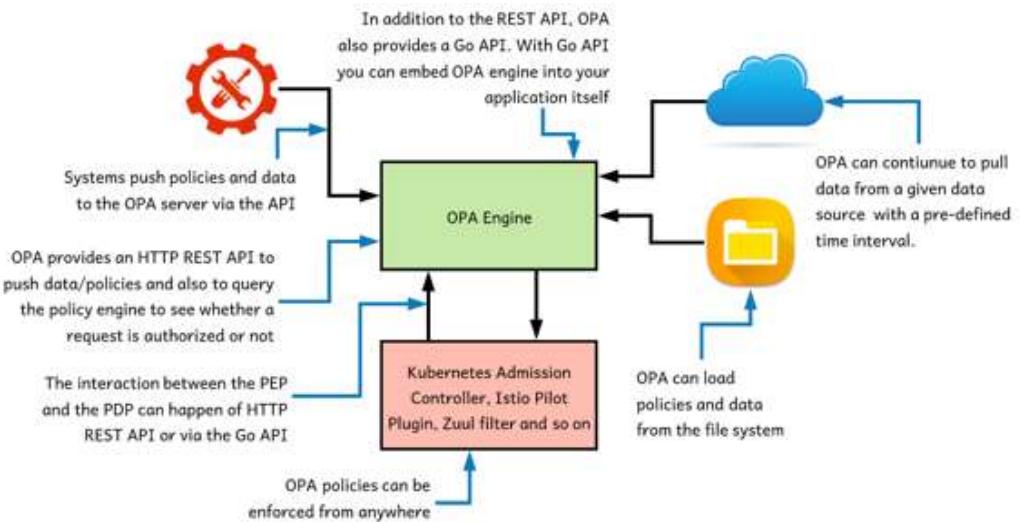


Figure G-2: An application or a policy enforcement point can integrate with OPA policy engine via its HTTP REST API or via the Go API.

When you run OPA server as a stand-alone deployment, it exposes a set of REST APIs that the policy enforcement point (PEP) can connect to check authorization. Here the OPA server acts as the policy decision point (PDP). The open source distribution of the OPA server does not come with a policy authoring tool or any user interface to create and publish policies to the OPA server. But you can use a tool like VS Code to create OPA policies and OPA has a plugin for VS Code. If you decide to embed OPA server (instead of using it as a hosted server) as a library to your application, you can use the Go API provided by OPA to interact with it.

Once you have the policies, you can use the OPA API to publish them to the OPA server. When you publish them via the API, OPA engine only keeps them in memory, and you need to build some mechanism to publish policies every time the server boots up. The other option is to copy the policy files to the file system and the OPA server will pick them up, at the time it boots up. Once again, if there are any policy changes you would need to restart the OPA server. There is an option to ask the OPA server to load policies dynamically from the file system, but that is not recommended in a production deployment. Also there is another option to push policies to the OPA server using a bundle server, which we discuss in detail under section G.7.

OPA has a policy information point (PIP) design to bring in external data to the PDP – or to the OPA server. This model is quite similar to the model we discussed before with respect to policies. Under section G.7 we discuss in detail how OPA brings in external data.

G.4 Deploying OPA as a Docker container

In this section we discuss how to deploy an OPA server as a Docker container. In OPA, there are multiple ways of loading policies. One way is to use the REST API provided by OPA. That's the approach we followed in chapter 5, under section 5.3. When we use the REST API, OPA keeps all the policy data in-memory, so when you restart the container again, then there should be way to push policies again. Even if you do not use Docker to run an OPA server, it's the same result. To avoid that, we can use Docker volume mounts (see appendix A). With volume mounts, we keep all the policies in a directory in the host file system – and then mount it to the OPA Docker container file system. If you look at the `appendix-g/sample01/run_opa.sh`, you will find the following Docker command. There we mount the `policies` directory from the current location of the host file system to the `policies` directory of the container file system under root.

```
docker run --mount type=bind,source="$(pwd)"/policies,target=/policies -p 8181:8181
openpolicyagent/opa:0.15.0 run /policies --server
```

To start the OPA server, run the following command from the `appendix-g/sample01` directory. This will load the OPA policies from the `appendix-g/sample01/policies` directory. In section G.6 we discuss OPA policies detail.

```
\> sh run_opa.sh
{"addrs":["":8181],"insecure_addr":"","level":"info","msg":"Initializing
server.", "time":"2019-11-05T07:19:34Z"}
```

You can use the following command from `appendix-g/sample01` directory to test the OPA server. The `appendix-g/sample01/policy_1_input_1.json` file carries the input data for the authorization request in JSON format. In section G.6 we discuss authorization requests in detail.

```
\> curl -v -X POST --data-binary @policy_1_input_1.json
      http://localhost:8181/v1/data/authz/orders

{"result":{"allow":true}}
```

The process of deploying OPA in Kubernetes is similar to deploying any other service on Kubernetes, as we discussed in appendix B. You can check OPA documentation available at <https://www.openpolicyagent.org/docs/latest/deployments/#kubernetes> for details.

G.5 Protecting OPA server with Mutual Transport Layer Security (mTLS)

In this section, we discuss how to protect the OPA server with mutual Transport Layer Security (mTLS). This will make sure all the communications happen between the OPA server and other client applications are encrypted. And also only the legitimate clients with proper

keys can talk to the OPA server. To protect the OPA server with mTLS we need to do following tasks.

- Generate a public/private key pair for the OPA server.
- Generate a public/private key pair for the OPA client.
- Generate a public/private key pair for the Certificate Authority (CA) client.
- Sign the public key of the OPA server with CA's private key to generate OPA server's public certificate.
- Sign the public key of the OPA client with CA's private key to generate OPA client's public certificate.

To perform all the above tasks we can use the `appendix-g/sample01/keys/gen-key.sh` script, with OpenSSL. Let's run the following Docker command from `appendix-g/sample01/keys` directory to spin up an OpenSSL Docker container. There you can see, we mount the current location (which is `appendix-g/sample01/keys`) from the host file system to the `/export` directory on the container file system

```
\> docker run -it -v $(pwd):/export prabath/openssl
#
```

Once the container booted up successfully, you'll find a command prompt where you can type OpenSSL commands. Lets run the following command to execute the `gen-key.sh` file, which will run a set of OpenSSL commands.

```
# sh /export/gen-key.sh
```

Once the above command executed successfully, you will find the keys corresponding to the certificate authority under `appendix-g/sample01/keys/ca` directory, the keys corresponding to the OPA server under `appendix-g/sample01/keys/opa` directory, and the keys corresponding to the OPA client under `appendix-g/sample01/keys/client` directory. If you want to understand the exact OpenSSL commands we ran during the key generation, please check appendix K. To start OPA server with TLS support, use the following command from `appendix-g/sample01` directory. In case you have already running OPA server, please stop it by pressing Ctrl+C on the corresponding command console.

```
\> sh run_opa_tls.sh
{"addr":["8181"],"insecure_addr":"","level":"info","msg":"Initializing
server.", "time":"2019-11-05T19:03:11Z"}
```

You can use the following command from `appendix-g/sample01` directory to test the OPA server. The `appendix-g/sample01/policy_1_input_1.json` file carries the input data for the authorization request in JSON format. Here we are using HTTPS to talk to the OPA server,

```
\> curl -v -X POST --data-binary @policy_1_input_1.json
      https://localhost:8181/v1/data/authz/orders
>{"result":{"allow":true}}
```

Let's check, what is in the `run_opa_tls.sh` script (listing G.1). The code annotations in listing G.1 explain what each argument means.

Listing G.1. Protecting OPA server endpoint with TLS

```
\> docker run \
    -v "$(pwd)"/policies:/policies \      #A
    -v "$(pwd)"/keys:/keys \               #B
    -p 8181:8181 \                      #C
    openpolicyagent/opa:0.15.0 \          #D
    run /policies \                     #E
    --tls-cert-file /keys/opa/opa.cert \ #F
    --tls-private-key-file /keys/opa/opa.key \ #G
    --server \                          #H
```

#A Instructs OPA server to load policies and from this directory
#B OPA server will find keys/certs for TLS from this directory
#C Port mapping
#D Name of the OPA Docker image
#E Run the OPA server by loading policies and data from the policies directory
#F Certificate used for TLS
#G Private key used for TLS
#H Start in the server mode

Now, the communication between the OPA server and the OPA client (cURL) is protected with TLS. But still, anyone having access to the OPA server's IP address can access it over TLS. There are two ways to protect the OPA endpoint for authentication: token authentication and mutual TLS. With token-based authentication the client has to pass an OAuth 2.0 token in the HTTP Authorization header as a bearer token - and also you need to write an authorization policy as explained in <https://www.openpolicyagent.org/docs/latest/security/>. In this section we only worry about securing the OPA endpoint with mTLS. To start the OPA server enabling mTLS, run the following command from `appendix-g/sample01` directory. In case you have already running the OPA server, please stop it by pressing `Ctrl+C` on the corresponding command console.

```
\> sh run_opa_mtls.sh
```

Let's check what is in the `run_opa_mtls.sh` script (listing G.2). The code annotations in listing G.2 explain what each argument means.

Listing G.2. Protecting OPA server endpoint with mTLS

```
\> docker run \
    -v "$(pwd)"/policies:/policies \
    -v "$(pwd)"/keys:/keys \
    -p 8181:8181 \
    openpolicyagent/opa:0.15.0 \
    run /policies \
    --tls-cert-file /keys/opa/opa.cert \
    --tls-private-key-file /keys/opa/opa.key \
    --tls-ca-cert-file /keys/ca/ca.cert \ #A
    --authentication=tls \ #B
```

```
--server
```

```
#A All the OPA clients must carry a certificate signed by this CA
#B Enables mTLS authentication
```

You can use the following command from appendix-g/sample01 directory to test the OPA server, which is now secured with mTLS. Here we are using HTTPS to talk to the OPA server along with the certificate and the key generated for the OPA client at the start of this section. The key and the certificate of the OPA client are available under appendix-g/sample01/keys/client directory.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @input_1_policy_1.json https://localhost:8181/v1/data/authz/orders/policy1
```

G.6 OPA policies

To define access control policies, OPA introduces a new programming language called, Rego. You can find more details about Rego from here: <https://www.openpolicyagent.org/docs/latest/policy-language/>. In this section we go through a set of OPA policies to understand the strength of the Rego programming language. All the policies we discuss under this section are available under the appendix-j/sample01/policies directory and already loaded into the OPA server we booted up in section G.5, which is protected with mTLS

Listing G.3. OPA policy written in Rego

```
package authz.orders.policy1    #A

default allow = false    #B

allow {    #C
  input.method = "POST"    #D
  input.path = "orders"
  input.role = "manager"
}

allow {
  input.method = "POST"
  input.path = ["orders",dept_id]    #E
  input.deptid = dept_id
  input.role = "dept_manager"
}
```

```
#A The package name of the policy. There cannot be two policies in the system with the same package name
#B All the requests by default are disallowed. If this is not set and no allowed rules are matched, then OPA will return
     undefined decision
#C Declares the conditions to allow access to the resource.
#D The value of the method parameter in the input document must be POST
#E The value of the path parameter in the input document must match this value, where the value of the dept_id is the
     deptid parameter from the input document
```

The policy defined in listing G.3 (`policy_1.rego`), has two **`allow`** rules. For an allow rule to return true, each statement within the allow block must return true. The first allow rule will return true only if a user with the `manager` role is trying to do an HTTP POST on the `orders` resource. The second allow rule will return true if a user with the `dept_manager` role is trying to do an HTTP POST on the `orders` resource under his/her own department. Let's try to evaluate this policy with two different input documents. First lets try with the input document in listing G.4 (`policy_1_input_1.json`). Run the following cURL command from `appendix-g/sample01` directory.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_1_input_1.json https://localhost:8181/v1/data/authz/orders/policy1
{"result":{"allow":true}}
```

Listing G.4. Input document with manager role

```
{
  "input":{
    "path":"orders",
    "method":"POST",
    "role":"manager"
  }
}
```

Let's try with another input document in listing G.5 (`policy_1_input_2.json`). Run the following cURL command from `appendix-g/sample01` directory. You can see how the response from OPA server changes by changing the values of the inputs.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_1_input_2.json https://localhost:8181/v1/data/authz/orders/policy1
{"result":{"allow":true}}
```

Listing G.5. Input document with dept_manager role

```
{
  "input":{
    "path":["orders",1000],
    "method":"POST",
    "deptid":1000,
    "role":"dept_manager"
  }
}
```

Now let's have a look at a slightly improved version of the policy in listing G.3. You can find this new policy in listing G.6 and its already deployed into the OPA server you are running. Here our expectation is that if a user is in the `manager` role, he/she will be able to do HTTP PUT, POST or DELETE on the `orders` resource and if a user is in the `dept_manager` role, he/she will be able to do HTTP PUT, POST or DELETE on `orders` resource under his/her own department. Also any user, irrespective of the role, should be able to do GET to the `orders`

resource under his/her own account. The annotations in the listing G.6 explain how the policy is constructed.

Listing G.6. OPA policy written in Rego

```
package authz.orders.policy2

default allow = false

allow {
    allowed_methods_for_manager[input.method]      #A
    input.path = "orders"
    input.role = "manager"
}

allow {
    allowed_methods_for_dept_manager[input.method]   #B
    input.deptid = dept_id
    input.path = ["orders",dept_id]
    input.role = "dept_manager"
}

allow {    #C
    input.method = "GET"
    input.empid = emp_id
    input.path = ["orders",emp_id]
}

allowed_methods_for_manager = {"POST","PUT","DELETE"}      #D
allowed_methods_for_dept_manager = {"POST","PUT","DELETE"}  #E
```

#A Checks whether the value of the method parameter is in the allowed_methods_for_manager array

#B Checks whether the value of the method parameter is in the allowed_methods_for_dept_manager array

#C Allows anyone to access orders resource under his/her own employ id.

#D The definition of the allowed_methods_for_manager array

#E The definition of the allowed_methods_for_dept_manager array

G.7 External data

During the policy evaluation, sometimes the OPA engine needs access to external data. In this section we discuss multiple approaches OPA provides to bring in external data for the policy evaluation. A detailed discussion on these different approaches is documented here: <https://www.openpolicyagent.org/docs/latest/external-data/>.

G.7.1 Push data

The push data approach to bring in external data to the OPA server uses the data API provided by the OPA server. Let's have a look at a simple example. This is in fact the same example we used in chapter 5 under the section 5.3. This policy in listing G.7 returns true, if method, path and the set of scopes coming in the input message match with some data read from an external data source, which is loaded under the package name data.order_policy_data.

Listing G.7. OPA policy using pushed external data

```
package authz.orders.policy3

import data.order_policy_data as policies      #A

default allow = false      #B

allow {      #C
    policy = policies[_]      #D
    policy.method = input.method      #E
    policy.path = input.path
    policy.scopes[_] = input.scopes[_]      #F
}
```

#A The package name of the policy.

#B Declares the set of statically registered data identified by order_policy

#C All the requests by default are disallowed. If this is not set and no allowed rules matched, then OPA will return undefined decision

#D Declares the conditions to allow access to the resource.

#E Iterate over values in the policies array

#F For an element in the policies array, check whether the value of the method parameter in the input, matches with the method element of policy

The policy in listing G.7 consumes all the external data from a JSON file (appendix-g/sample01/order_policy_data.json), which we need to push to the OPA server using the OPA data API. Assuming your OPA server is running on port 8181, you can use the following curl command from the appendix-g/sample01 directory to publish data to the OPA server. Please keep in mind that here we are only pushing external data, not the policy. The policy that consumes the data is already in the OPA server, which you can find in the appendix-g/sample01/policies/policy_3.rego file.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -H "Content-Type: application/json" -X PUT --data-binary @order_policy_data.json https://localhost:8181/v1/data/order_policy_data
```

Listing G.8. A set of resources in Order Processing microservice, defined as OPA data

```
[
{
  "id": "r1",      #A
  "path": "orders",      #B
  "method": "POST",      #C
  "scopes": ["create_order"]      #D
},
{
  "id": "r2",
  "path": "orders",
  "method": "GET",
  "scopes": ["retrieve_orders"]
},
{
  "id": "r3",
  "path": "orders/{order_id}",
  "method": "PUT",
```

```

    "scopes": ["update_order"]
}
]

```

#A An identifier for the resource path.
#B The resource path.
#C The HTTP method.
#D To do an HTTP POST to the order resource, you must have this scope.

Now you can use the following cURL command with the input message (appendixg/sample01/policy_3_input_1.json) from listing G.9 to check if the request is authorized or not.

```

\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_3_input_1.json https://localhost:8181/v1/data/authz/orders/policy3
{"result":{"allow":true}}

```

Listing G.9. OPA input document

```

{
  "input": {
    "path": "orders",
    "method": "GET",
    "scopes": ["retrieve_orders"]
  }
}

```

With the push data approach, you have the control on when you want to push the data to the OPA server. Whenever the external data gets updated, you can push the updated data to the OPA server. However, the push data based approach has its own limitations too. When you use the data API to push external data into the OPA server, OPA server keeps the data in cache (in-memory), and whenever you restart the server, you need to push the data again.

G.7.2 Loading data from the file system

In this section we discuss how to load external data from the file system. When we start the OPA server, we need to specify from which directory on the file system, the OPA server should load data files along with the policies. Let's have a look at the appendix-g/sample-01/run_opa_mtls.sh shell script (listing G.10). The code annotations explain how OPA loads policies from the file system at the start up.

Listing G.10. Protecting OPA server endpoint with mTLS

```

docker run \
  -v "$(pwd)"/policies:/policies \
  -v "$(pwd)"/keys:/keys \
  -p 8181:8181 \
  openpolicyagent/opa:0.15.0 \
  run /policies \
  --tls-cert-file /keys/opa/opa.cert \
  --tls-private-key-file /keys/opa/opa.key \
  --tls-ca-cert-file /keys/ca/ca.cert \

```

```
--authentication=tls \
--server
```

#A Doing a volume mount so that OPA sever can load data and policies from this directory
#B Run the OPA server by loading policies and data from the policies directory

The OPA server you already have running has the policy and the data we are going to discuss in this section. Let's first check the external data file (`order_policy_data_from_file.json`), which is available under `appendix-j/sample01/policies` directory. This is the same file you saw in listing G.6 except there is a slight change to the structure of the file. You can find the updated data file in listing G.11.

Listing G.11. OPA policy using external data that comes with the request

```
{
  "order_policy_data_from_file": [
    {
      "id": "p1",
      "path": "orders",
      "method": "POST",
      "scopes": ["create_order"]
    },
    {
      "id": "p2",
      "path": "orders",
      "method": "GET",
      "scopes": ["retrieve_orders"]
    },
    {
      "id": "p3",
      "path": "orders/{order_id}",
      "method": "PUT",
      "scopes": ["update_order"]
    }
  ]
}
```

In listing G.11 you can see, in the JSON payload, we use a root element called, `order_policy_data_from_file`. OPA server derives the package name corresponding to this data set as `data.order_policy_data_from_file`, which is used in the policy in listing G.12. The policy in listing G.12 is exactly the same as you saw in listing G.8 except the package name has changed.

Listing G.12. OPA policy using pushed external data

```
package authz.orders.polic4

import data.order_policy_data_from_file as policies

default allow = false

allow {
  policy = policies[_]
  policy.method = input.method
  policy.path = input.path
```

```
policy.scopes[_] = input.scopes[_]
```

Now you can use the following cURL command with the input message (appendix-g/sample01/policy_4_input_1.json) from listing G.9 to check if the request is authorized or not.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_4_input_1.json https://localhost:8181/v1/data/authz/orders/policy4
{"result":{"allow":true}}
```

One issue with loading data from the file system is, when there is any update, you need to restart the OPA server. There is a configuration option (see appendixg/sample01/run_opa_mtls_watch.sh) to ask OPA server to load policies dynamically (without a restart), but that option is not recommended for production deployments. In practice, if you deploy an OPA server in a Kubernetes environment, then you can keep all your policies and data in a git repo and use an init container along with the OPA server in the same pod to pull all the policies and data from the git repo during the bootup of the corresponding pod. This process is same as the approach we discussed in chapter 11, under the section 11.2.7 to load key stores. Whenever there is an update to the policies or data, we need to restart the pods.

G.7.3 Overload

The overload approach to bring in external data to the OPA server uses the input document itself. When the policy enforcement point (PEP) builds the authorization request, it can embed external data into the request itself. Say for example, the `orders` API knows for anyone to do an HTTP POST to it; you need to have the `create_order` scope. Rather pre-provisioning all the data into the OPA server, the PEP can send them along with the authorization request. Let's have a look at a slightly modified version of the policy in listing G.7. You can find the updated policy in listing G.13.

Listing G.13. OPA policy using external data that comes with the request

```
package authz.orders.policy5

import input.external as policy

default allow = false

allow {
    policy.method = input.method
    policy.path = input.path
    policy.scopes[_] = input.scopes[_]
}
```

In listing G.13, you can see that we use the `input.external` package name to load the external data from the input document. Let's have a look at the input document in listing G.14, which carries the external data with it.

Listing G.14. OPA policy using external data that comes with the request

```
{
    "input": {
        "path": "orders",
        "method": "GET",
        "scopes": ["retrieve_orders"],
        "external": {
            "id": "r2",
            "path": "orders",
            "method": "GET",
            "scopes": ["retrieve_orders"]
        }
    }
}
```

Now you can use the following cURL command with the input message (appendix-g/sample01/policy_5_input_1.json) from listing G.14 to check the request is authorized or not.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_5_input_1.json https://localhost:8181/v1/data/authz/orders/policy5
{"result":{"allow":true}}
```

Reading external data from the input document does not work all the time. For example, there should be a trust relationship between the OPA client (or the policy enforcement point) and the OPA server. In section G.7.4 we discuss a better way of sending data in the input document, in a more reliable way.

G.7.4 JSON Web Token (JWT)

JSON Web Token (JWT) provides a reliable way of transferring data over the wire between multiple parties in a cryptographically secure way. If you are new to JWT, please check appendix H. OPA provides a way to pass a JWT in the input document. The OPA server can verify the JWT and then read data from it. Let's go through an example. First we need to have a security token service (STS), which issues a JWT. You can spin up an STS using the following command. This is the same STS we discussed in chapter 10.

```
\> docker run -p 8443:8443 prabath/insecure-sts-ch10:v1
```

The STS will start on port 8443. Once it got started run the following command to get a JWT.

```
\> curl -v -X POST --basic -u applicationid:applicationsecret -H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" -k -d "grant_type=password&username=peter&password=peter123&scope=foo" https://localhost:8443/oauth/token
```

In this command, applicationid is the client ID of the web application and applicationsecret is the client secret (which are hardcoded in the STS). If everything

works fine, the security token service returns an OAuth 2.0 access token, which is a JWT (or a JWS, to be precise):

```
{"access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiJE1NTezMTIzNzYsInVzZXJfbmFtZSI6InBldGVyIiwiYXV0aG9yaXRpZXMiOlSiUk9MRV9VU0VSI10sImp0aSI6IjRkMmJiNjQ4LTQ2MWQtNGV1Yy1hZT1jLTIVLYWUxZjA4ZTJhMiIsImNsawVudF9pZC16ImFwcGxpY2F0aw9uaWQiLCJzY29wZSI6WyJmb28iXX0.tr4yUmGLtsH7q9Ge2i7gxyTs0Oa0RS0Yoc2uBuAW50VKZcVsIITWV3bDN0FVBzimpAPy33tvicFROhBf0vThqKXzzG00SkURN5bnQ4uFLAP0NpZ6BuDjvVmwxNxrQp21VXl41Q4eTvuyZozjUSCxzCI1Lnw5EFFi22J73g1_mRm2j-dEhBp1TvMaRKLBDk2hzIDVku5oj_g0DBFm3aLS-IJjYoCimIm2igcesXkhipRjtjNcrjSegBbGgyXHVak2gB7I07ryVwl_Re5yX4sV9x6xNwCxc_DgP9hHLzPM8yz_K97j1T6Rr1XZB1veyjfks_XIXgu5qizRm9mt5xg", "token_type": "bearer", "refresh_token": "", "expires_in": 5999, "scope": "foo", "jti": "4d2bb648-461d-4eec-ae9c-5eae1f08e2a2"}
```

Now you can extract out the JWT from the above output, which is the value of the `access_token` parameter. It's bit lengthy and please make sure that you copy the complete string. In listing G.15 you can find the input document and there we use the copied value of the JWT as the value of the `token` parameter. Listing G.15 shows only a part of the JWT and you can find the complete input document in appendix-g/sample01/policy_6_input_1.json file.

Listing G.15. OPA policy using external data that comes with the request

```
{
  "input": {
    "path": ["orders", 101],
    "method": "GET",
    "empid": 101,
    "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9... "
  }
}
```

Let's have a look at the policy corresponding to the input document in listing G.15, which you can find in listing G.16. The code annotations in listing G.16 explain all key instructions.

Listing G.16. OPA policy using external data that comes with the request

```
package authz.orders.policy6

default allow = false

certificate = `-----BEGIN CERTIFICATE----- #A
MIICzCCAa+gAwIBAgIEHP9VkjAN...
-----END CERTIFICATE-----` #B

allow {
  input.method = "GET"
  input.empid = emp_id
  input.path = ["orders", emp_id]
  token.payload.authorities[_] = "ROLE_USER"
}

token = {"payload": payload} {
  io.jwt.verify_rs256(input.token, certificate) #B
  [header, payload, signature] := io.jwt.decode(input.token) #C
  payload.exp >= now_in_seconds #D
}
```

```

}

now_in_seconds = time.now_ns() / 1000000000 #E

#A The PEM encoded certificate to validate the JWT.
#B Verify the signature of the JWT – following the RSA SHA256 algorithm
#C Decode the JWT
#D Check whether the JWT is expired
#E Find the current time in seconds. The now_ns() returns time in nanoseconds

```

Now you can use the following cURL command with the input message (appendix-g/sample01/policy_6_input_1.json) from listing G.15 to check the request is authorized or not.

```
\> curl -k -v --key keys/client/client.key --cert keys/client/client.cert -X POST --data-binary @policy_6_input_1.json https://localhost:8181/v1/data/authz/orders/policy6
{"result":{"allow":true}}
```

In listing G.16 to do the JWT validation, we first validate the signature and then check the expiration. OPA has a built-in function called `io.jwt.decode_verify(string, constraints)`, which validates all in one go. You can use this function to validate the signature, expiration (exp), not before use (nbf), audience, issuer and so on. However, at the time of this writing, this function was not working as expected and you can track the progress of this issue related to this from <https://github.com/open-policy-agent/opa/issues/1882>. Also, you can find all the OPA functions available to verify JWT from here: <https://www.openpolicyagent.org/docs/latest/policy-reference/#token-verification>.

G.7.5 Bundle API

To bring in external data to an OPA server under the bundle API approach, first you need to have a bundle server. A bundle server is an endpoint, which hosts a bundle. For example, the bundle server can be an AWS S3 bucket or a GitHub repository. A bundle is a gzipped tarball, which carries OPA policies and data files under a well-defined directory structure. You can find more details on how to create a bundle from here: <https://www.openpolicyagent.org/docs/latest/management/#bundles>.

Once the bundle endpoint is available, you need to update the OPA configuration file with the bundle endpoint, the credentials to access the bundle endpoint (if it is secured), the polling interval and so on and pass the configuration file as a parameter when you spin up the OPA server. More details on these configuration options are documented here: <https://www.openpolicyagent.org/docs/latest/configuration/>. Once the OPA server is up, it will continuously poll the bundle API to get the latest bundle after each predefined time interval.

If your data changes very frequently, then you will find some drawbacks in using the bundle API. Since the OPA server polls the bundle API after a predefined time interval, if you frequently update the policies or data, then there can be a case you make authorization decisions based on some stale data. To fix that, you can reduce the polling time interval, but then again, that will increase the load on the bundle API.

G.7.6 Pull Data during Evaluation

At the time of this writing, pull data during evaluation approach is an experimental feature. With this approach you do not need to load all the external data into the OPA server's memory, rather you pull data as and when needed during the policy evaluation. To implement pull data during evaluation, you need to use the OPA built-in function, `http.send`. So, you would need to host an API (or a microservice) over HTTP (which is accessible to the OPA server) to accept data requests from the OPA server and respond back with the corresponding data. More details on how to use `http.send` is documented here with some examples: <https://www.openpolicyagent.org/docs/latest/policy-reference/#http>.

G.8 OPA integrations

As we discussed early in the chapter, OPA is a general-purpose policy engine. Being a general-purpose policy engine, it can address a large variety of access control use cases. In this section we briefly discuss three use cases, which are related to a microservices deployment.

G.8.1 Istio mixer plugin

Istio is a service mesh implementation developed by Google, Lyft and IBM. It is open source and the most popular service mesh implementation at the time of this writing. If you are new to Istio or service mesh architecture please check appendix C. Istio introduces a component called Mixer, which runs on Istio control plane (figure G.3). Mixer takes care of precondition checking, quota management and telemetry reporting. For example, when a request hits the Envoy proxy at the data plane, it will talk to the Mixer API to do precondition checking to see whether it's okay to proceed with that request or not. Mixer has a rich plugin architecture, so you can chain multiple plugins in the precondition check phase. For example, you can have a mixer plugin, which can connect to an external policy decision point (PDP) to evaluate a set of access control policies against the incoming request.

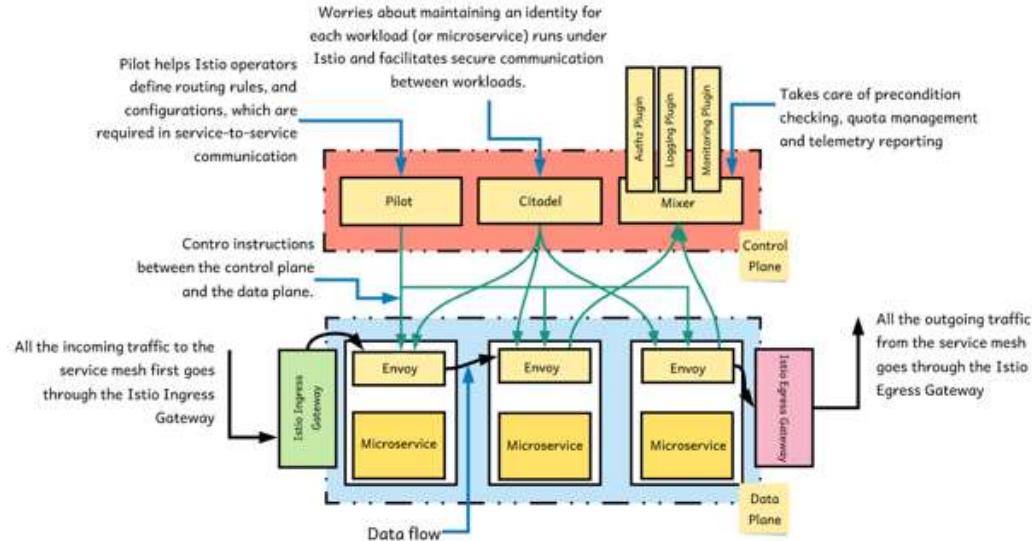


Figure G.3 Istio high-level architecture with a control plane and a data plane.

Istio integrates with OPA via the OPA Mixer adapter. When a request hits the Envoy proxy in the data plane it, it does a check API call to the Mixer. This API call carries certain attributes with respect to the request, for example path, headers and so on. Then Mixer hands over the control to the OPA mixer adapter. The OPA mixer adapter, which embeds OPA engine as an embedded library, does the authorization check against the defined policies and returns back the decision to Mixer, and then to the Envoy proxy. You can find more details on the OPA Mixer plugin from here: <https://github.com/istio/istio/tree/master/mixer/adapter/opa>.

G.8.2 Kubernetes admission controller

The Kubernetes admission control is a component that's run in the Kubernetes API server. In appendix B, under the section B.18, we discuss how Kubernetes internal communication works and the role of an admission controller. When an API request arrives at the Kubernetes API server, it goes through a set of authentication and authorization plugins (figure G.4). The authenticated and the authorized request goes through another set of plugins called admission controller plugins.

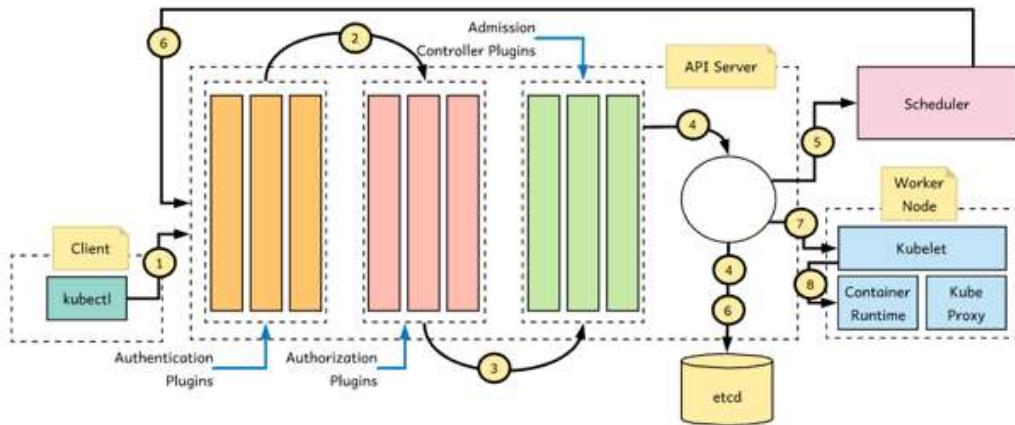


Figure G.4 A request generated by kubectl passes through authentication, authorization and admission controller plugins of the API server, validated and then stored in etcd. Scheduler and kubelet respond to events generated by the API server.

To authorize all the requests coming into the Kubernetes API server, you can use OPA Gatekeeper, which is a Kubernetes admission controller. You can find more details on OPA Gatekeeper from here: <https://github.com/open-policy-agent/gatekeeper>. Also how to deploy OPA Gatekeeper on Kubernetes to do Kubernetes ingress validation is documented at <https://www.openpolicyagent.org/docs/latest/kubernetes-tutorial/>.

G.8.3 Apache Kafka

In chapter 9 we discussed Kafka under the context of securing reactive microservices. Apache Kafka is the most popular message broker implementation used in microservice deployments. To use OPA for Kafka authorization, you need to engage the OPA Authorizer plugin with Kafka. To authorize a request, the OPA Authorizer plugin talks to a remote OPA server over HTTP. You can find more details on OPA Kafka Authorizer from here: https://github.com/open-policy-agent/contrib/tree/master/kafka_authorizer. In a Kubernetes deployment, you would deploy the OPA server as sidecar along with Kafka on the same pod.

G.9 OPA alternatives

Since OPA was introduced in 2016, OPA is now becoming the de facto implementation of fine-grained access control, mostly in the microservices domain. There are a couple of alternatives to OPA, but at the time of this writing, none of them are as popular as OPA.

eXtensible Access Control Markup Language (XACML) is an open standard developed by Organization for the Advancement of Structured Information Standards (OASIS). The XACML standard introduces a policy language based on XML and a schema based on XML for authorization requests and responses. OASIS released XACML 1.0 specification in 2003 and at

the time of this writing the latest is XACML 3.0. XACML was popular many years back, but over the time, as with the popularity of XML based standards went down, XACML adoption too went down rapidly. Also, XACML as a policy language is quite complex, though very powerful. If you are looking for an open source implementation of XACML 3.0, please check the Balana project, which is available at <https://github.com/wso2/balana>.

Speedle is another open source alternative to OPA, which is also a general purpose authorization engine. Speedle is developed by Oracle and relatively new. Its' too early to comment on how Speedle competes with OPA and at the time of this writing Oracle cloud internally uses Speedle. You can find more details on Speedle from here: <https://speedle.io/>.

H

JSON Web Token (JWT)

We've discussed JSON Web Token (JWT) many times in this book. In chapter 2, we talked about how you can use a JWT as an OAuth 2.0 self-contained access token, and in chapter 4, we described how OpenID Connect uses a JWT as its ID token to transfer user claims from the OpenID provider to the client application. In chapter 7, we discussed how to pass end-user context in a JWT among services in a microservices deployment. In chapter 11, we examined how each pod in Kubernetes uses a JWT to authenticate to the Kubernetes API server. In chapter 12, we showed how an Istio service mesh uses JWT to verify the end user context at the Envoy proxy. Finally, in appendix G, we described how an Open Policy Agent (OPA) uses JWT to carry policy data along with the authorization request. All in all, JWT is an essential ingredient in securing a microservices deployment. In this appendix, we discuss JWT in detail.

H.1 What is a JSON Web Token (JWT)?

A *JWT* (pronounced *jot*) is a container that carries different types of assertions or claims from one place to another in a cryptographically safe manner. An *assertion* is a strong statement about someone or something issued by some entity. Imagine that your state's Department of Motor Vehicles (DMV) can create a JWT (to represent your driver's license) with your personal information, which includes your name, address, eye color, hair color, gender, date of birth, license expiration date, and the license number. All these items are attributes or claims about you and are also known as *attribute assertions*. The DMV is the issuer of the JWT. Anyone who gets this JWT can decide whether to accept what's in it as true, based on the level of trust they have in the issuer of the token (in this case, the DMV). But before accepting a JWT, how do you know who issued it? The issuer of a JWT signs it using the issuer's private key. In this scenario, a bartender, who represents the recipient of the JWT, can verify the signature of the JWT and see who signed it (see figure H.1).

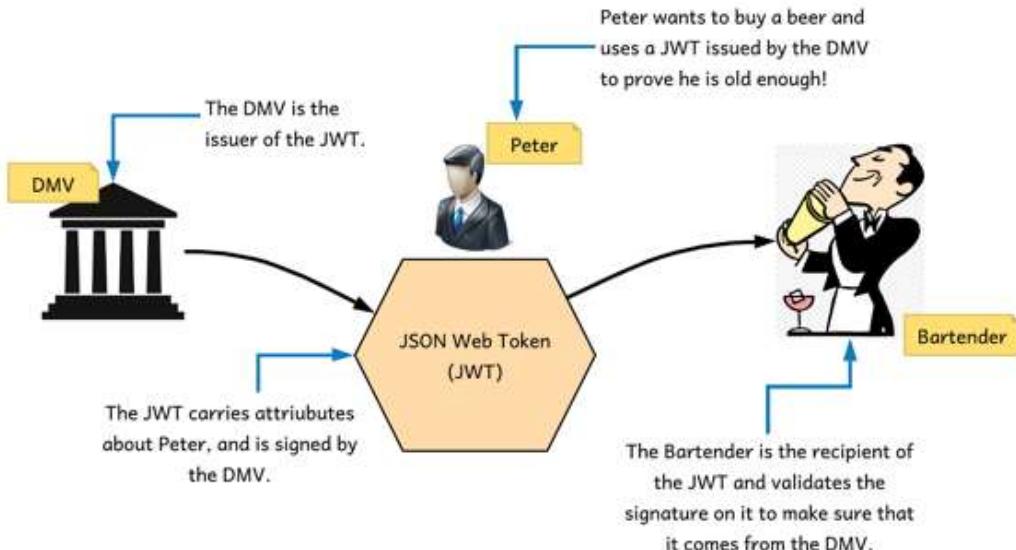


Figure H.1 JWT is used as a container to transport assertions from one place to another in a cryptographically safe manner. The bartender, who is the recipient of the JWT, accepts the JWT only if he or she trusts the DMV or the issuer of the JWT.

In addition to attribute assertions, a JWT can carry authentication and authorization assertions. In fact, a JWT is a container; you can fill it with anything you need. An authentication assertion might be the username of the user and how the issuer authenticates the user before obtaining the assertion. In the DMV use case, an authentication assertion might be your first name and last name (or even your driver's license number), or how you are known to the DMV.

An authorization assertion is about the user's entitlements or what the user can do. Based on the assertions the JWT brings from the issuer, the recipient can decide how to act. In the DMV example, if the DMV decides to embed the user's age as an attribute in the JWT, that data is an attribute assertion, and a bartender can do the math to calculate whether the user is old enough to buy a beer. Also, without sharing the user's age with a bartender, the DMV may decide to include an authorization assertion stating that the user is old enough to buy a beer. In that case, a bartender will accept the JWT and let the user buy a beer. He wouldn't know user's age, but the DMV authorized the user to buy beer.

In addition to carrying a set of assertions about the user, JWT plays another role behind the scenes. Apart from the end user's identity, JWT also carries the issuer's identity, which is the DMV in this case. The issuer's identity is implicitly embedded in the signature of the JWT. By looking at the corresponding public key when validating the token, the recipient can figure out who the token issuer is.

H.2 What does a JWT look like?

Before we delve deep into the JWT use cases within a microservices deployment, take a closer look at a JWT. Figure H.2 shows the most common form of a JWT. This figure may look like gibberish unless your brain is trained to decode base64url-encoded strings.

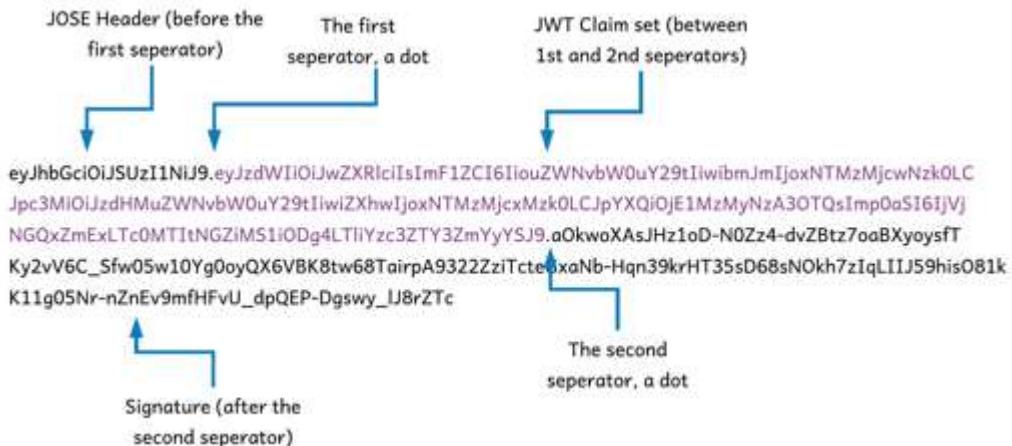


Figure H.2 A base64url-encoded JWT formatted as a JSON Web Signature (JWS)

The JSON Web Signature (JWS) is the most commonly used format of a JWT, which has three parts with a dot (.) separating each part.

- The first part is known as the JSON Object Signing and Encryption (JOSE) header.
- The second part is the claim set or the body (or payload).
- The third part is the signature.

The JOSE header expresses the metadata related to the token, such as an algorithm used to sign the message. Here's the base64url-decoded JOSE header:

```
{
  "alg": "RS256",
}
```

The JWT claim set carries the assertions (between the first and second separators). Following is the base64url-decoded claim set:

```
{
  "sub": "peter",
  "aud": "*_ecomm.com",
  "nbf": 1533270794,
  "iss": "sts.ecomm.com",
  "exp": 1533271394,
  "iat": 1533270794,
  "jti": "5c4d1fa1-7412-4fb1-b888-9bc77e67ff2a"
```

```
}
```

We don't intend to go through every attribute ID in this JWT claim set—only a few important ones: `iat`, `exp`, `nbf`, `iss`, `aud`.

H.2.1 JWT expiration and issued time

In this section, we discuss two important attribute IDs the JWT claim set carries: the `exp` attribute and the `iat` attribute. The value of the `exp` attribute in the decoded JWT claim set (shown in figure H.3) expresses the time of expiration in seconds, which is calculated from 1970-01-01T0:0:0Z as measured in Universal Time Coordinated (UTC). Any recipient of a JWT must make sure that the time represented by the `exp` attribute is not in the past when accepting a JWT. The `iat` attribute in the JWT claim set expresses the time when the JWT was issued. That too is expressed in seconds and calculated from 1970-01-01T0:0:0Z as measured in UTC.

The difference between `iat` and `exp` in seconds isn't the lifetime of the JWT all the time when there's a `nbf` (not before) attribute. You shouldn't start processing a JWT (or accept it as a valid token) before the time specified in a `nbf` attribute. The value of `nbf` is also expressed in seconds and calculated from 1970-01-01T0:0:0Z as measured in UTC.

H.2.2 An issuer of a JWT

Another key attribute ID in this JWT claim set is the `iss` attribute. The `iss` attribute ID is set to the value `sts.ecomm.com` in the decoded JWT claim set shown in figure H.3, which is the issuer of the JWT. The JWT is signed by the issuer's private key. In a typical microservices deployment within a given trust domain, all the microservices trust a single issuer. The issuer is also called a security token service (STS).

```
{
  "sub": "peter",
  "aud": "* .ecomm.com",
  "nbf": 1533270794,
  "iss": "sts.ecomm.com",
  "exp": 1533271394,
  "iat": 1533270794,
  "jti": "5c4d1fa1-7412-4fb1-b888-9bc77e67ff2a"
}
```

Figure H.3 Base64url-decoded JWT claim set with highlighted claims.

H.2.3 The audience of a JWT

Another key attribute ID in this JWT claim set is the `aud` attribute. The `aud` attribute specifies the audience of the token—in other words, the intended recipient of the token. In figure H.3, it's set to the string value `*.ecomm.com`. The value of the `aud` attribute can be any string or an URI that's known to the microservice or the recipient of the JWT. Each microservice must check the value of the `aud` parameter to see whether it's known before accepting any JWT as valid. If you've a microservice called `foo` with the audience value `foo.ecomm.com`, the microservice should reject any JWT carrying the `aud` value `bar.ecomm.com`, for example. The logic in accepting or rejecting a JWT based on audience is up to the corresponding microservice and to the overall microservices security design. By design, you can agree that any microservice will accept a token with the audience value `<microservice identifier>.ecomm.com` or `*.ecomm.com`, for example.

H.3 JSON Web Signature (JWS)

The JWT explained in section H.2 (and, as a reminder, shown in figure H.4) is also a JSON Web Signature (JWS). JWS is a way to represent a signed message. This message can be anything, such as a JSON payload, an XML payload, or a binary.

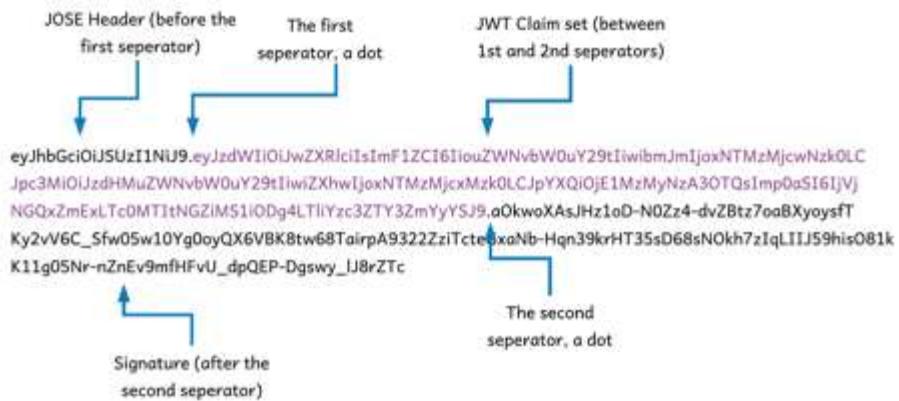


Figure H.4 Base64url-encoded JWT formatted as a JSON Web Signature (JWS)

A JWS can be serialized in two formats or represented in two ways: compact serialization and JSON serialization. A JWT is a base64url-encoded, compact-serialized JWS with three parts (figure H.5). Section H.3 details what each of these components mean.



Figure H.5 A JWT that is a compact, serialized JWS with a JOSE header, a claim set, and a signature.

With JSON serialization, the JWS is represented as a JSON payload (see figure H.6). It's not called a JWT. The `payload` parameter in the JSON-serialized JWS can carry any value. The message being signed and represented in figure H.6 is a JSON message with all its related metadata.

```
{
  "payload": "eyJpc3MiOiJqb2UiLCJleHAiOiEzMjAwMTkzODI",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": {
        "kid": "2014-06-29"
      },
      "signature": "cC4hiUPoj9Eetdgtv3hF80EGrhvB"
    },
    {
      "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header": {
        "kid": "e909097a-ce81-4036-9562-d21d2992db0d"
      },
      "signature": "DtEhU3ljbEg8L38VWAfUAq0yKAM"
    }
  ]
}
```

Figure H.6 A JWS with JSON serialization that includes related metadata

Let's see how to use the open source Nimbus (<https://connect2id.com/products/nimbus-jose-jwt>) Java library to create a JWS. The source code related to all the examples used in this

chapter is available in the <https://github.com/microservices-security-in-action/samples> GitHub repository, inside the appendix-h directory.

NOTE Before running the samples in this appendix, make sure that you have downloaded and installed all the required software as mentioned in section 2.1.1.

Let's build the sample, which builds the JWS, and run it. Run the following Maven command from the appendix-h/sample01 directory. It may take a couple of minutes to finish the build process when you run this command for the first time. If everything goes well, you should see the BUILD SUCCESS message at the end:

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

Now run your Java program to create a JWS with the following command (from the appendix-h/sample01/lib directory). If it executes successfully, it prints the base64url-encoded JWS:

```
\> java -cp ..//target/com.manning.mss.appendixh.sample01-1.0.0.jar:*
com.manning.mss.appendixh.sample01.RSASHA256JWTBuilder
eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJwXRlcIisImF1ZCI6IiouZWNVbW0uY29tIiwibmJmIjoxNTMzMjcwNzk0LCJp
c3MiOiJzdHMuZWNVbW0uY29tIiwizXhwIjoxNTMzMjcxMzk0LCJpYXQiOjE1MzMyNzA3OTQsImp0aSI6IjvNGQ
xZmExLTc0MTITNGZiMS1iODg4LTliYzc3TY3ZmYyYSJ9.aOkwoXAsJH21oD-N0Zz4-
dvZBtz7oaBXyoysFTKy2vV6C_Sfw05w10Yg0oyQX6VBK8tw68TairpA9322ZziTcteGxaNb-
Hqn3krHT35sD68sN0kh7zIqLIIJ59his081kk11g05Nr-nZnEv9mfHFvU_dpQEP-Dgswy_lJ8rZTc
```

You can decode this JWS by using the JWT decoder available at <https://jwt.io>. Following is the decoded JWS claim set or payload:

```
{
  "sub": "peter",
  "aud": "* ecommerce.com",
  "nbf": 1533270794,
  "iss": "sts.ecomm.com",
  "exp": 1533271394,
  "iat": 1533270794,
  "jti": "5c4d1fa1-7412-4fb1-b888-9bc77e67ff2a"
}
```

NOTE If you get any errors while executing the previous command, check whether you executed the command from the correct location. It has to be from inside the appendix-h/sample01/lib directory, not from the appendixh/sample01 directory.

Take a look at the code that generated the JWT. It's straightforward and self-explanatory with comments. You can find the complete source code in the sample01/src/main/java/com/manning/mss/appendixh/sample01/RSASHA256JWTBuilder.java file.

The following method does the core work of JWT generation. It accepts the token issuer's private key as an input parameter and uses it to sign the JWT with RSA-SHA256 (listing H.1):

Listing H.1. The content of the RSASHA256JWTBuilder.java file

```
public static String buildRsaSha256SignedJWT(PrivateKey privateKey) throws JOSEException {
    // build audience restriction list.
    List<String> aud = new ArrayList<String>();
    aud.add("*.ecomm.com");

    Date currentTime = new Date();

    // create a claim set.
    JWTClaimsSet jwtClaims = new JWTClaimsSet.Builder().
        issuer("sts.ecomm.com").

        // set the subject value - JWT belongs to this subject.
        subject("peter").

        // set values for audience restriction.
        audience(aud).

        // expiration time set to 10 minutes.
        expirationTime(new Date(new Date().getTime() + 1000 * 60 * 10))..

        // set the valid from time to current time.
        notBeforeTime(currentTime).

        // set issued time to current time.
        issueTime(currentTime).

        // set a generated UUID as the JWT identifier.
        jwtID(UUID.randomUUID().toString()).build();

    // create JWS header with RSA-SHA256 algorithm.
    JWSHeader jswHeader = new JWSHeader(JWSAlgorithm.RS256);

    // create signer with the RSA private key..
    JWSSigner signer = new RSASSASigner((RSAPrivateKey) privateKey);

    // create the signed JWT with the JWS header and the JWT body.
    SignedJWT signedJWT = new SignedJWT(jswHeader, jwtClaims);

    // sign the JWT with HMAC-SHA256.
    signedJWT.sign(signer);

    // serialize into base64-encoded text.
    String jwtInText = signedJWT.serialize();

    // print the value of the JWT.
    System.out.println(jwtInText);

    return jwtInText;
}
```

H.4 JSON Web Encryption (JWE)

In the preceding section, we stated that a JWT is a compact-serialized JWS. It's also a compact-serialized JWE. Like JWS, a JWE represents an encrypted message using compact serialization or JSON serialization. A JWE is called a JWT only when compact serialization is used. In other words, a JWT can be either a JWS or a JWE, which is compact serialized and used to carry different types of assertions from one place to another. JWS addresses the integrity and no repudiation aspects of the data contained in it, while JWE protects the data for confidentiality.

A compact-serialized JWE (see figure H.7) has five parts; the parts are base64url-encoded and separated by dots (.). A JOSE header is the part of the JWE that carries metadata related to the encryption. The JWE encrypted key, initialization vector, and authentication tag are related to the cryptographic operations performed during the encryption. We won't talk about those in detail here. If you're interested, we recommend the blog "JWT, JWS, and JWE for Not So Dummies" at <https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3>. Finally, the ciphertext part of the JWE includes the encrypted text.



Figure H.7 A JWT that's a compact-serialized JWE.

With JSON serialization, the JWE is represented as a JSON payload. It isn't called a JWT. The payload parameter in the JSON-serialized JWE can carry any value as discussed previously. The actual message is encrypted and represented in figure H.8 as a JSON message with all related metadata.

```
{
  "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "unprotected": [
    "jku": "https://server.example.com/keys.jwks"
  ],
  "recipients": [
    {
      "header": {
        "alg": "RSA1_5",
        "kid": "2011-04-29"
      },
      "encrypted_key": "UGhi0guC7iuEvf_NPVaXsCMoL0mwvc1GyqlI9X5hH59_i8J0PH5ZZyNfGy2xGd"
    },
    {
      "header": {
        "alg": "A128KW",
        "kid": "7"
      },
      "encrypted_key": "6KB707dM9YTigHtLvtgHQ8mKwboJW3af9loc1zkDTHzBC21lrT1o0Q"
    }
  ],
  "iv": "AxYB0CtDaGlsbGljb3RoZQ",
  "ciphertext": "KDlTTXchhZTGufMYn0YG54HffxPSUrfmqCHxaI9wOGY",
  "tag": "Mz-VPPyU4RlcuYv1IwIvw"
}
```

Figure H.8 A JWE with JSON serialization and all related metadata.

Let's see how to use the open-source Nimbus Java library to create a JWE. The source code related to all the samples used in this chapter is available in the <https://github.com/microservices-security-in-action/samples> Git repository inside the appendix-h directory. Before you delve deep into the Java code that you'll use to build the JWE, try to build the sample and run it. Run the following Maven command from the appendix-h/sample02 directory. If everything goes well, you should see the BUILD SUCCESS message at the end:

```
\> mvn clean install
[INFO] BUILD SUCCESS
```

Now run your Java program to create a JWE with the following command (from the appendix-h/sample02/lib directory). If it executes successfully, it prints the base64url-encoded JWE:

```
\> java -cp ..../target/com.manning.mss.appendixh.sample02-1.0.0.jar:*
com.manning.mss.appendixh.sample02.RSAOAEPJWTBuilder
eyJlbmMiOiJBMTI4R0NNIwiYWxnIjoiUlNBLU9BRVAifQ.Cd0KjNwSbq50PxcJQ1ESValmRGPF7BFUNpqZFFKTcd-
9XAmVE-
zOTsnv78SikT0K8fuwszHDnz2eONUahbg8eR9oxDi9kmXaHeKXyZ9Kq4vhg7WJPJXSUonwGxcibgECJySEJxZaT
```

```
mA1E_8pUaiU6k5UHvxPUDtE0pnN5XD82cs.0b4jWQHFbBaM_azM.XmwvMBzrLcNW-
oBhAfMozJ1mESfG6o96WT958B0yfjpGmmbdJdIjirjCBTUATdOPkLg6-
YmPsitaFm7pFAUsHkm4_K1ZrE5HuP43VM0gBXSe-
41dDDNs7D2nZ5QfpeoYH7zQNocCjbseJPFPEw311nBRfjzNoDEzvKMsxhgCZNLTv-
tpKh6mKIXXYxdxVoBcIXN90UYi.mVLD4t-85qcTiY8q3J-kmg
```

Following is the decrypted JWE payload:

```
JWE Header:{ "enc": "A128GCM", "alg": "RSA-OAEP" }
JWE Content Encryption Key: Cd0KjNwSbq50PxcJQ1ESValmRGPF7BFUNpqZFFKTCd-9XAmVE-
zOTsnv78SikTOK8fuwszHDnz2e0NUahbg8eR9oxDi9kmXaHeKXyZ9Kq4vhg7WJPJXSUonwGxcibgECJySEJxZaT
mA1E_8pUaiU6k5UHvxPUDtE0pnN5XD82cs
Initialization Vector: 0b4jWQHFbBaM_azM
Ciphertext : XmwvMBzrLcNW-oBhAfMozJ1mESfG6o96WT958B0yfjpGmmbdJdIjirjCBTUATdOPkLg6-
YmPsitaFm7pFAUsHkm4_K1ZrE5HuP43VM0gBXSe-
41dDDNs7D2nZ5QfpeoYH7zQNocCjbseJPFPEw311nBRfjzNoDEzvKMsxhgCZNLTv-
tpKh6mKIXXYxdxVoBcIXN90UYi
Authentication Tag: mVLD4t-85qcTiY8q3J-kmg
Decrypted Payload:
{ "sub": "peter", "aud": "* .ecomm .com", "nbf": 1533273878, "iss": "sts .ecomm .com", "exp": 1533274
478, "iat": 1533273878, "jti": "17dc2461-d87a-42c9-9546-e42a23d1e4d5" }
```

NOTE If you get any errors while executing the previous command, check whether you executed the command from correct location. It has to be from inside the appendix-h/sample02/lib directory, not from the appendix-h/sample02 directory.

Now take a look at the code that generated the JWE. It's straightforward and self-explanatory with code comments. You can find the complete source code in the sample02/src/main/java/com/manning/mss/appendixh/sample02/RSAOAEPJWTBuilder.java file. The following method does the core work of JWE encryption. It accepts the token recipient public key as an input parameter and uses it to encrypt the JWE with RSA-OAEP.

Listing H.2. The content of the RSAOAEPJWTBuilder.java file

```
public static String buildEncryptedJWT(PublicKey publicKey) throws JOSEException {
    // build audience restriction list.
    List<String> aud = new ArrayList<String>();
    aud.add("*.ecomm.com");

    Date currentTime = new Date();

    // create a claim set.
    JWTClaimsSet jwtClaims = new JWTClaimsSet.Builder()

        // set the value of the issuer.
        issuer("sts.ecomm.com").

        // set the subject value - JWT belongs to this subject.
        subject("peter").

        // set values for audience restriction.
        audience(aud).

        // expiration time set to 10 minutes.
        expiresAt(new Date(System.currentTimeMillis() + 10 * 60 * 1000));
}
```

```
expirationTime(new Date(new Date().getTime() + 1000 * 60 * 10)).  
  
// set the valid from time to current time.  
notBeforeTime(currentTime).  
  
// set issued time to current time.  
issueTime(currentTime).  
  
// set a generated UUID as the JWT identifier.  
jwtID(UUID.randomUUID().toString()).build();  
  
// create JWE header with RSA-OAEP and AES/GCM.  
JWEHeader jweHeader = new JWEHeader(JWEAlgorithm.RSA_OAEP,  
                                     EncryptionMethod.A128GCM);  
  
// create encrypter with the RSA public key.  
JWEEncrypter encrypter = new RSAEncrypter((RSAPublicKey) publicKey);  
  
// create the encrypted JWT with the JWE header and the JWT payload.  
EncryptedJWT encryptedJWT = new EncryptedJWT(jweHeader, jwtClaims);  
  
// encrypt the JWT.  
encryptedJWT.encrypt(encrypter);  
  
// serialize into base64-encoded text.  
String jwtInText = encryptedJWT.serialize();  
  
// print the value of the JWT.  
System.out.println(jwtInText);  
  
return jwtInText;  
}
```



Secure Production Identity Framework For Everyone

In chapter 6, we detailed the challenges in key management, including key provisioning, trust bootstrapping, certificate revocation, key rotation, and key usage monitoring. In a typical microservices deployment, each microservice is provisioned with a key pair. In chapter 6 you did that by manually copying Java keystore files to the Order Processing and Inventory microservices. Doing things manually won't work in a microservices deployment, however—everything must be automated.

Ideally, during the continuous integration/continuous delivery (CI/CD) pipeline, the keys should be generated and provisioned to the microservices. When the keys are provisioned to all the nodes/microservices, the next challenge is to bootstrap trust between nodes. Why would a node trust a request initiated from another node? That's the trust bootstrap problem we need to solve. Then again, just like provisioning keys to each microservice, the provisioned keys must be rotated before they expire. In this appendix, we discuss how SPIFFE (Secure Production Identity Framework For Everyone) helps to address key provisioning, trust bootstrapping, and key rotation problems.

I.1 What is SPIFFE?

SPIFFE is an open standard that defines a way that a microservice (a workload in SPIFFE terminology) can establish an identity. SPIFFE Runtime Environment (SPIRE) is an open-source reference implementation of SPIFFE. While helping to establish an identity for each microservice in a given deployment, SPIFFE also solves the trust bootstrap problem. In this section, we discuss how SPIFFE works.

The inspiration behind SPIFFE came from three projects at Netflix, Facebook, and Google. Metatron is the Netflix project, which we've already discussed in chapter 6. It solves the

credential-provisioning problem by injecting long-lived credentials into each microservice during the continuous delivery phase. Facebook has an internal public key infrastructure (PKI) project, which helps bootstrapping trust among systems that are secured with mutual Transport Layer Security (mTLS). Google has a project called Low Overhead Authentication Services (LOAS), which is a cryptographic-key distribution system that helps establish an identity for all the jobs running on the Google infrastructure.

SPIFFE not only solves the trust bootstrap problem but also provides node attestation. It provisions keys to a given workload (or microservice) if the corresponding attestation policies are satisfied. Another benefit of SPIFFE is that the keys provisioned to the nodes never leave them, and there's no need to have any long-lived credentials provisioned to the nodes (or services) during a continuous delivery phase. SPIFFE doesn't worry about certificate revocation; it relies on short-lived certificates and takes care of key rotation.

I.2 How SPIFFE/SPIRE works?

As we discussed in section I.1, SPIFFE is a standard or specification, whereas SPIFFE is the corresponding reference implementation. The SPIFFE architecture has two main components: the SPIFFE agent (also known as the node agent) and the SPIFFE server. The SPIFFE agent runs on the same node where the workload (or microservice) is running. If you run your microservice on an Amazon EC2 machine, for example, the SPIFFE agent runs on the same EC2 node. If you run your microservice in a Docker container, the SPIFFE agent runs on the same host machine that runs the Docker container. In other words, the SPIRE agent shares an OS kernel with the workload. The following list walks you through the steps defined in figure I.1 to explain how SPIFFE works and sets forth its design principles:

1. The SPIFFE node agent authenticates to the SPIFFE server.

Authentication happens via a component called a *node attester*. A node attester runs on both the SPIRE server and the SPIFFE node agent. If the workload is running on an Amazon EC2 node, for example, the node attester at the SPIFFE node agent's end picks the corresponding AWS instance identity document

(<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-identity-documents.html>) and passes it to the SPIFFE server for authentication. The AWS instance identity document is a document signed by Amazon that includes the metadata related to the corresponding EC2 instance. Within a given EC2 instance, you can get only the AWS instance identity document corresponding to that node. When the document is passed to the SPIFFE server, the AWS node attester at the server side validates the signature of the document by using Amazon's public key.

The node attester is an extension point. If your workload runs in a Kubernetes environment (we talk about Kubernetes in appendix B), the node agent that runs in a Kubernetes node (not along with a microservice in the same pod) uses a JSON Web Token (JWT) provisioned to it by the Kubernetes cluster to prove its identity to the SPIFFE server. This JWT uniquely identifies the node.

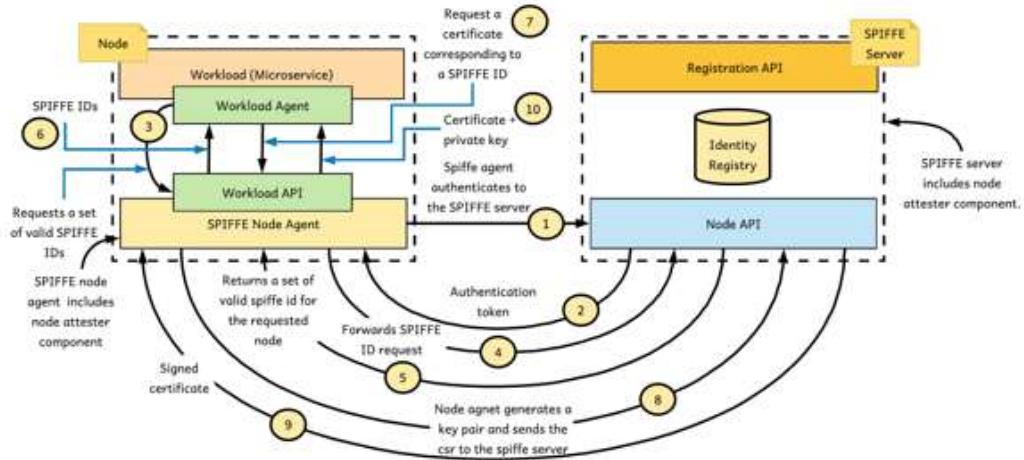


Figure I.1 Communication between the SPIFFE node agent and the SPIFFE server

2. When the node is authenticated properly, the SPIFFE server issues a set of SPIFFE IDs corresponding to the node on which the SPIFFE agent is running, along with a set of selectors. A selector defines the criteria to associate a SPIFFE ID with a node.

The SPIFFE ID (<https://github.com/spiffe/spiffe/blob/master/standards/SPIFFE-ID.md>) is a URI in the format spiffe://trust-domain/workload-identifier. A SPIRE server runs in each trust domain. An example SPIFFE ID is spiffe://foo.com/retail/order-processing, in which foo.com reflects the trust domain, and retail/order-processing represents the workload.

The process completed with steps 1 and 2 is known as *node attestation*. A registry in the SPIRE server keeps a set of attestation policies that defines the criteria under which a given SPIFFE ID can be assigned to a workload. The policy shown in figure I.1 (the registry entry) says that the SPIFFE ID spiffe://foo.com/retail/order-processing can be issued only to a node running under AWS security group sg-e566bb82 and a workload with the UID 1002. You can define a workload in several other ways, but typically, you use at least two mechanisms: one at the infrastructure level (such as a security group) and the other at the workload level (UID).

3. Each workload (or microservice) can optionally have a SPIFFE workload agent, which knows how to talk to the workload API exposed by the SPIFFE node agent.

In a Kubernetes deployment this workload agent runs in the same pod along with the corresponding microservice (or the workload). The workload agent (on behalf of the workload) talks to the SPIFFE node agent and asks for its identity. The workload API is node-local. In the UNIX operating system, for example, it's exposed via UNIX domain

sockets.¹ The workload agent doesn't need to know where it's running, such as Amazon EC2, Google Cloud, Kubernetes, or any other platform. It simply asks the workload API, "Who am I?"

4. The SPIFFE node agent validates the request coming from the workload agent and tries to identify it.

If the workload API is exposed via UNIX domain sockets, the SPIRE agent can find the metadata related to the workload via the OS kernel. This way, the SPIRE agent can find out the UID and PID related to the workload and can scan all the selectors to find a match. If a match is found, that match is the SPIFFE ID related to this workload. If you're on Kubernetes (see appendix B), the SPIFFE agent can also talk to a kubelet to find out whether the PID related to the workload is scheduled on that node. If all goes well, the SPIRE agent generates a key pair for that workload, creates a Certificate Signing Request (CSR), and sends the CSR to the SPIFFE server.

5. The SPIFFE server validates the CSR and issues a SPIFFE Verifiable Identity Document (SVID) to the SPIRE agent.

The SVID has three basic components: a SPIFFE ID, a public key that represents the workload, and a valid signature by the SPIRE server. A SVID can be in multiple formats; it can be an X509 certificate (<https://github.com/spiffe/spiffe/blob/master/standards/X509-SVID.md>) or a JWT (<https://github.com/spiffe/spiffe/blob/master/standards/JWT-SVID.md>). In case of an X509 SVID, what you get is a signed certificate corresponding to the CSR that the SPIRE agent submitted.

The SPIFFE server is acting as a certificate authority (CA), and all the issued certificates must have the corresponding SPIFFE ID in the X509 Subject Alternate Name (SAN) field. The SPIFFE server can also delegate the certificate-issuing part to an upstream CA. After the certificate is issued to the corresponding workload via the SPIRE agent, the SPIFFE server updates all the other workloads with this SVID so that all of those trust this workload. Also, what's returned to the SPIFFE agent in step 5 isn't the SVID alone; it's also a certificate bundle corresponding to the workload, which this workload can trust as well.

¹ A UNIX domain socket is a data communications endpoint for exchanging data between processes executing on the same host operating system.

6. The SPIFFE node agent passes the signed certificate (SVID) and corresponding private key to the workload, along with the certificate bundle returned from the SPIRE server. Now the workload can use its key pair to talk to other workloads over mTLS.

J

gRPC fundamentals

gRPC (<https://grpc.io/>) is an open source remote procedure call framework (or a library), originally developed by Google. It's in fact the next generation of a system called Stubby, which Google has been using internally within Google for over a decade. gRPC achieves efficiency for communication between systems using HTTP/2 as the transport and protocol buffers as the interface definition language (IDL). In chapter 8 we discussed how to secure communications between microservices over gRPC. In this appendix we discuss the fundamentals of gRPC. If you're interested in reading more about gRPC, we recommend *Practical gRPC* by Joshua Humphries, David Konsumer, David Muto, Robert Ross and Carles Sistare (Bleeding Edge Press, 2018) or gRPC: Up and Running by Kasun Indrasiri and Danesh Kuruppu (O'Reilly Media, 2020).

J.1 What is gRPC?

Many of us are familiar with functions in computer programs. A function is a named section in a program that performs a specific task. A software program usually has a main function, which is called by the underlying operating system when the program starts to run. A function in a typical program is invoked/executed by another function (or the main function) running within the same program. RPC stands for **Remote Procedure Call**. As its name implies RPC is a protocol whereby a program can execute a function that is running on a remote host/computer on the network. RPC typically involves generating some method stubs at the client side that makes it look like a function invocation is local but is actually remote. See following example.

```
Registry registry = LocateRegistry.getRegistry(serverIP, serverPort);
Products products = (Products) registry.lookup(name);
int count = products.getCount();
```

In this example, the object `products` although is a local variable, its `getCount()` method does a remote procedure call over the network to a method running on a remote server identified by `serverIP` and `serverPort`. The `getCount()` method on the server is where the actual business logic of the function resides. The method on the client application is simply a surrogate for the same method on the server application. Figure J.1 illustrates how the client application uses a stub to communicate with the server application.

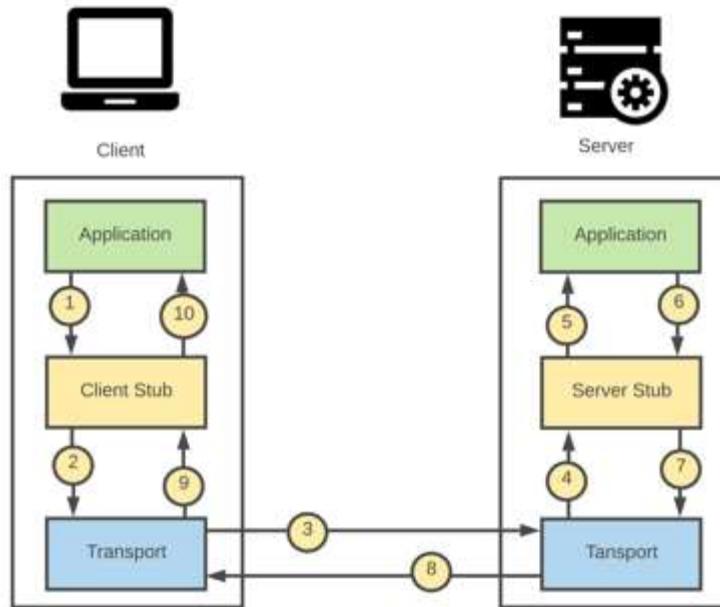


Figure J.1 When communicating over RPC the client and server both use stubs to interface with each other.

gRPC has now become the method of choice for communications that happen between microservices. This is primarily because of the performance optimizations it offers compared to other common mechanisms such as JSON over HTTP. As mentioned in previous chapters, a microservice-driven application has many interactions that happen between microservices over the network. Therefore, whatever optimizations we can achieve at the network layer are realized in several orders of magnitude in real world applications of it. Figure J.2 shows interactions between microservices to complete a given user operation.

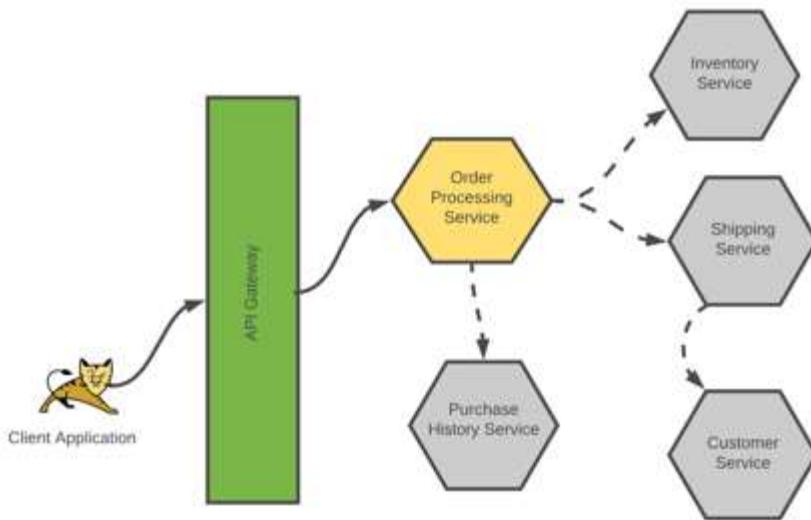


Figure J.2 – In a typical microservices architecture a single user operation results in many network interactions that happen between various microservices.

As you can see in figure J.2, when a user places an order many interactions happen between various microservices. The Order Processing microservice talks to the Inventory microservice to update the stock information. It also talks to the Shipping microservice for delivery. The Shipping microservice talks to the Customer microservice for getting delivery information. The Order Processing microservice also updates the customer purchase history by talking to the Customer microservice.

There are four interactions that happen between various microservices to serve a single **order** operation requested by a user. In this particular use case, since there are four interactions, happen between different microservices, whatever the benefits we gain over JSON/HTTP (measured by time) is realized by an order of magnitude of 4. Similarly, the advantages we gain with much larger applications, which may have hundreds of microservices, become much more significant. gRPC is better performing for microservices compared to JSON/XML over HTTP for two primary reasons.

- gRPC uses Protocol Buffers, also known as protobuf.
- gRPC uses of HTTP/2 transport protocol as opposed to HTTP/1.1.

J.2 Understanding Protocol Buffers

In this section we introduce Protocol Buffers and explain how they have been essential in the development of gRPC. We also talk about the benefits provide in terms of efficiency in data transfer.

When using JSON over HTTP for communicating messages between clients and servers the whole JSON message is transmitted in plaintext form. The payload is repetitive and sometimes unnecessary. This is because formats such as JSON/XML have been designed to be human readable. But, in practice only machines process these messages. While JSON/XML formats makes it easier to understand the message structures being passed along the network, when it comes to the application runtime this isn't necessarily important. **Protocol Buffers are a flexible, efficient, and automated mechanism for serializing structured data.** You can think of it as JSON or XML but with following exceptions.

- With much smaller sizes for representing the same messages.
- With much shorter time durations for processing those messages.
- Much simpler to understand, given its resemblance to programming languages.

Google created Protocol Buffers in 2001 to deal with an index server request response protocol. It was publicly released in 2008. The current version of the language is version 3, this is known as proto3. The examples we are following in this chapter are in this version. With Protocol Buffers, you first need to define how your data needs to be structured. These structures are defined in files having a `.proto` extension format. Listing J.1 shows what a simple `.proto` file looks like. It defines how a simple `Customer` object can be defined.

Listing J.1 A simple .proto file

```
syntax = "proto3";

message Customer {
    string name = 1;
    int32 id = 2;
    string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        string number = 1;
        PhoneType type = 2;
    }

    repeated PhoneNumber phone = 4;
}
```

As you may observe from listing J.1, each message type consists of one or more unique typed (string, int) fields. Each field has a unique number to identify it. Each typed field in a message has a default value. Following are the defaults by each type.

Table J.1 Default values for message types

| Type | Default value |
|-------------------------|--|
| strings | An empty string |
| numbers (numeric types) | Zero |
| boolean | False |
| bytes | Empty |
| enums | The first enum in the list, which should be set to 0 |
| message types | Depends on the programming language |

You can also organize your data structure in a hierarchical fashion by using message types within other message types, similar to how we have used the `PhoneNumber` message within the `Customer` message in listing J.1. The protobuf compiler then uses these structures to auto generate source code that can be used to read data from streams and populate these structures, and also to convert data in these structures to streams. The generated code can be made available in a variety of programming languages supported by the protobuf compiler.

Let's look at a quick sample to understand what this code generation looks like. Before we begin, check out the samples for this appendix from <https://github.com/microservices-security-in-action/samples/tree/master/appendix-j/>. We are going to generate Java code from a proto file using the spring-boot grpc module. The following sample has been tested on Java versions 8 and 11. You need to have Java 8+ and Maven version 3.2+ installed on your machine to try this out. Navigate to the `appendix-j/sample01` directory using your command line tool and execute the command below.

```
\> mvn compile
```

You should see a success message if the compilation is successful, and a target directory being created as well. Navigate to the newly created `target/generated-sources/protobuf/java/com/manning/mss/appendixj/sample01` directory. You should see a file named `Customer.java`. If you inspect its methods you will notice that it has functions such as `getName()`, `setName()` and so on which perform the manipulation of the fields we have declared in the proto file. The Java programs we will implement will use these auto-generated

functions to exchange data between clients and servers using gRPC. The compiled form of the above class can be found in the `target/classes` directory.

Let's also look at the `proto` file we just compiled. Open the `appendix-j/sample01/src/main/proto/customer.proto` file using a text editor or an IDE. The `syntax = "proto3";` statement right at the top of the file instructs the compiler that we are using protobuf version 3. The package statement `com.manning.mss.appendixj.sample01;` specifies the Java package name that should be included in the Java code being auto generated. You should see the same package statement in the generated `Customer.java` file. The option `java_multiple_files = true;` statement instructs the compiler to generate separate Java source files for each parent message type. In this example, we have one parent message type (`Customer`) only. However, this statement becomes handy when we have multiple messages to build code for since it neatly breaks down the source into multiple files instead of one large file.

J.3 Understanding HTTP/2 and its benefits over HTTP/1.x

In this section we talk about HTTP/2 and discuss how it has benefitted gRPC to become much more performant compared to JSON/XML over HTTP. One main reason for gRPC's growth in popularity is the performance gains it provides compared to similar alternatives such as JSON over HTTP. gRPC uses HTTP/2 as its transport layer protocol. HTTP/2 provides request multiplexing and header compression which increases its performance significantly. It also employs binary encoding of frames which makes the data being transferred much more compact and efficient for processing. Let's take a closer look at Request Response Multiplexing and Binary Framing.

J.3.1 Request response multiplexing and its performance benefits

In this section we introduce the concept of request multiplexing, which is used in the HTTP/2 protocol for efficient data exchange between communicating parties. We first introduce the problem in HTTP/1.x, and then look at how request multiplexing solves that problem.

In a client-server communication happening over HTTP/1.x, if the client wants to make multiple requests to the server in parallel to improve performance, multiple TCP¹ connections have to be used. This is a consequence of the HTTP/1.x delivery model where responses are sequential. By default, HTTP/1.x requests that happen over a single TCP connection are sequential as well. The protocol allows a client to send multiple requests to the server on a

¹TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

single TCP connection, using HTTP pipelining², but it involves lots of complexity and has been known to cause a lot of problems. It is therefore rarely in use; sequential requests are the default. Irrespective of whether the client application uses HTTP pipelining or not, only a single response is allowed to be sent back from the server at a given time on a single TCP connection. This can cause lots of inefficiencies, which forces applications using HTTP/1.x to use multiple TCP connections even for requesting data from a single host. Figure J.3 illustrates a scenario where HTTP pipelining is in use to make parallel requests to a server over a single TCP connection and shows the sequential nature of responses being sent back.

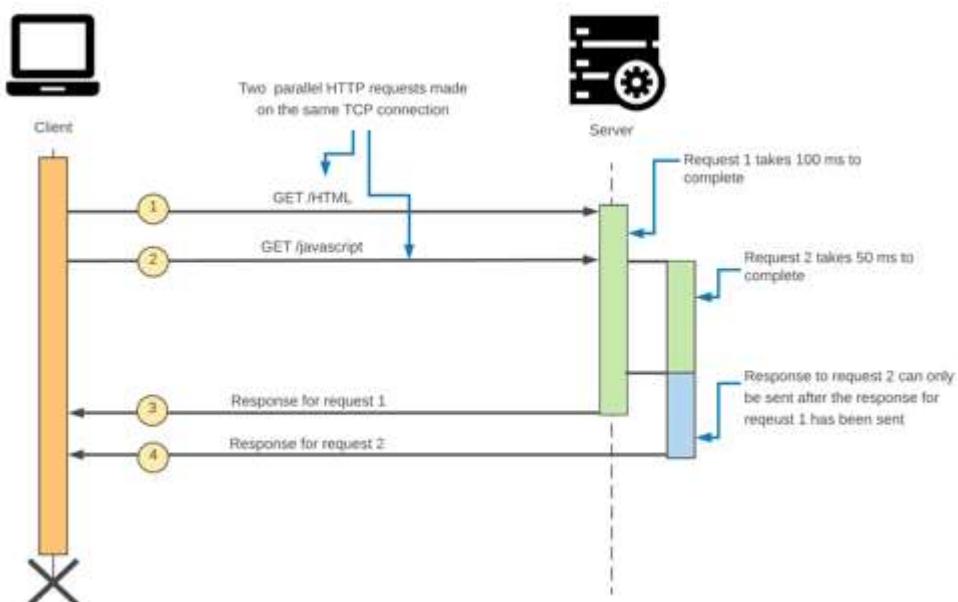


Figure J.3 – A client application making two parallel requests to the server over a single TCP connection. The server processes the requests in parallel. Even though the server completes processing the second request first, it needs to wait until the response to the first request is sent before sending the response to the second request.

As you can see from figure J.3, a client application makes two parallel requests to the server over a single TCP connection to render a web page. The server processes the GET /HTML

² Pipelining is the process a client sends successive requests over a single persistent TCP connection to a server without waiting for responses.

request first and `GET /javascript` request next. Preparing the first response takes 100 milliseconds (ms) and preparing the second response takes 50ms. Given the nature of the HTTP/1.x protocol, responses must be delivered to the client in sequential order. Therefore, even though the server completes preparing the second response much earlier than preparing the first response, it needs to wait until the first response is sent before the second response can be sent. This causes the client application to wait longer than is ideal before it can render the full web page it requested. This problem is also known as the **head-of-line blocking problem**. As we mentioned earlier, this limitation has forced client applications to use multiple TCP connections in parallel. Figure J.4 illustrates how client applications work around the head-of-line blocking problem by using multiple TCP connections in parallel.

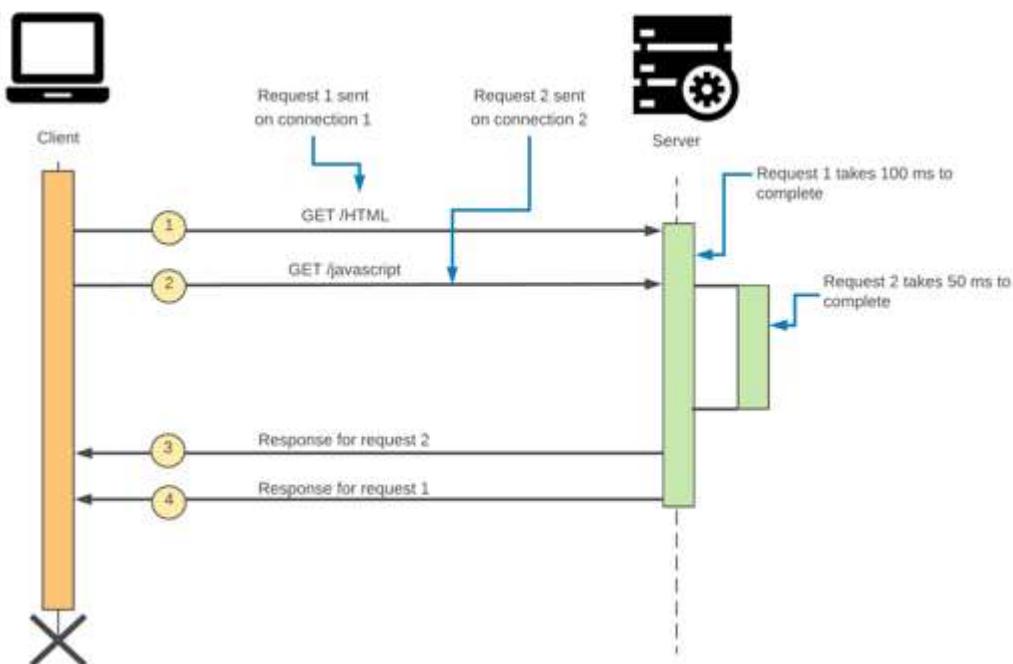


Figure J.4 – A client application making two parallel requests to the server on two distinct TCP connections. The server processes the requests in parallel. Responses to requests are sent back to the client in the order of request completion.

As illustrated in figure J.4, the `GET /HTML` request and `GET /javascript` requests are sent from the client to the server using two different TCP connections. Since the request with lower overhead (`GET /javascript`) completes first, the server can now send back the response to it without waiting for the other request to complete. This allows the client application to start

rendering the web page much earlier than in the previous case, where only a single TCP connection was used.

Using multiple concurrent TCP connections may sound like the solution to the head-of-line blocking problem. However, when being applied in practice, there is a limit on the number of TCP connections that can be created between communicating parties. This is mainly due to the resource limitations such as CPU, file I/O, and network bandwidth. A web browser would typically create a maximum of six concurrent TCP connections to a given host (web domain). Therefore, in the context of a web browser, the maximum level of concurrency we can achieve is six. All communication within a given single TCP connection is still sequential.

This is where request and response multiplexing in the HTTP/2 protocol becomes useful. The binary framing layer in HTTP/2 removes the aforementioned limitation in HTTP/1.x by allowing an HTTP message to be broken down into individual frames, interleaved, and then reassembled on the other side. Let's take a look at figure J.5 for a better understanding of this capability.

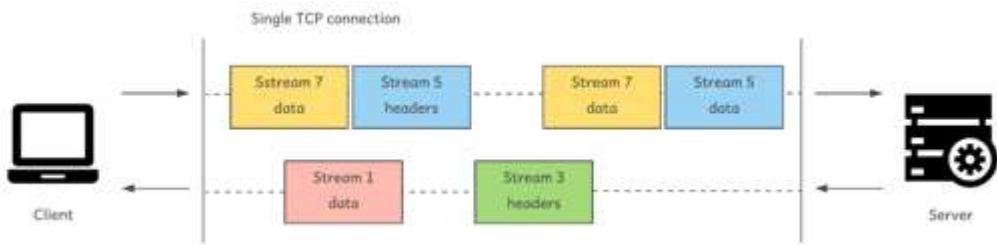


Figure J.5 – A client and server communicating using the HTTP/2 protocol. The requests and responses are multiplexed over a single TCP connection so that multiple messages can be transmitted concurrently without a message having to block over another message.

As you can see from figure J.5, with the HTTP/2 protocol, we can transmit multiple messages concurrently. The sending party breaks down each HTTP message into multiple frames of different types (DATA frames, HEADER frames and so on) and assigns them to a stream. The receiving party reassembles the messages based on the streams and starts processing each message as soon as each message completes reassembly. This gets rid of the head-of-line blocking problem we discussed under HTTP/1.x. The multiplexing capability in HTTP/2 gives us numerous benefits compared to HTTP/1.x as listed below.

- Interleaving of multiple requests in parallel without blocking on any one.
- Interleaving of multiple responses in parallel without blocking on any one.
- Using a single TCP connection between client and server, which reduces our resource utilization massively and also reduces operational costs.
- Improving the efficiency of client applications and servers by reducing idle time waiting on one another.
- Avoiding underusing our network bandwidth and improving the application efficiency.

Binary Framing and Streaming are the two fundamental concepts that allow HTTP/2 to multiplex requests and responses. Let us take a brief look at what they are and how they have helped the HTTP/2 protocol.

J.3.2 Understanding binary framing and streams in HTTP/2

In this section we look at the fundamental differences in how messages are encoded and exchanged between the HTTP/1.x vs. HTTP/2 protocols. We discuss in brief the concepts of binary framing and how frames get assigned to streams to allow multiplexing of requests and responses.

HTTP messages are composed of textual information. As the name HTTP itself implies (Hyper “**Text**” Transfer Protocol), it includes textual information that is encoded in ASCII and spans over multiple lines with newline delimiters included. With HTTP/1.x, these messages were openly transmitted over the network. However, with HTTP/2, each message is now divided into HTTP frames³. See figure 8.6 to understand how an HTTP message is usually broken down into frames.

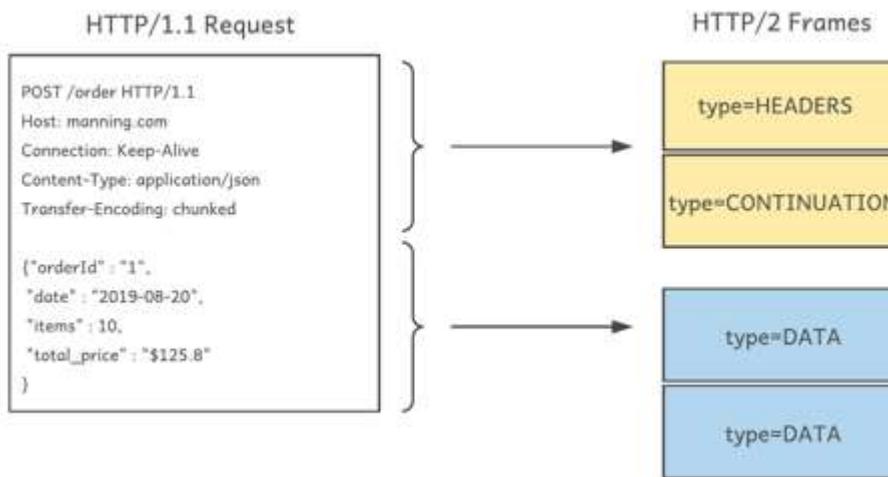


Figure J.6 – An HTTP/1.x message is broken down into multiple frames. The first chunk of headers is put into a frame typed HEADERS and the consequent header chunks are put into frames typed CONTINUATION. The request body is broken down into frames typed DATA.

³A frame is the smallest unit of communication that carries a specific type of data. For example, HTTP headers, message payload, and so on.

As shown in figure J.6, an HTTP message is broken down into multiple frames. Each frame has a type associated with it, which helps the receiver of the frame interpret the data in it accordingly. HTTP headers are transmitted in a frame typed HEADERS. Consequent headers of the same sequence are transmitted in a frame typed CONTINUATION. The request payload is transmitted in a frame typed DATA. A frame can hold a maximum of 16 megabytes of data. The HTTP/2 standards set the size of DATA frames to 16 kilobytes by default and allow the communicating parties to negotiate on higher values if necessary.

When initiating a communication channel a set of events take place, as listed below.

1. The client first breaks down the request message into binary frames and then assigns the stream ID of the request to the frames. This way each frame containing the binary data of the particular request gets associated to a single stream.
 2. The client then initiates a TCP connection with the server and starts sending the frames over this connection.
 3. Once the server receives the frames, it starts assembling them to form the request message, then starts processing the request.
 4. Once the server is ready to respond back to the client, the server breaks down the response into frames and assigns them the same stream ID as the request frames.
- Although frames can be transmitted in parallel on a single TCP connection, the stream ID in each frame allows the receiver to identify the proper message each frame belongs to. This scenario is illustrated in figure 8.5.

You may have noticed in figure J.5 that all stream IDs in the figure were odd numbers. This didn't happen coincidentally. The HTTP/2 protocol supports bi-directional streaming, which we talk about later in the chapter as well. This basically means that the client and server can both initiate the transmission of frames, unlike in HTTP/1.x, where only the client can initiate a transmission to the server. Client-initiated frames are assigned to streams with odd-numbered IDs and server-initiated frames are assigned to even-numbered stream IDs. This prevents the possibility of the client and server both initiating a stream with the same ID. The occurrence of such a scenario would have made it impossible for the receiver to properly identify the message a particular frame belongs to.

J.4 The different types of RPC available in gRPC

In this section, we look at the following different types of RPC available in the gRPC protocol and the types of scenarios in which each one of them become useful.

- Channels
- Metadata
- Unary RPC
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC

J.4.1 Understanding channels

A gRPC channel represents a connection made from a client application to a host and port on a remote gRPC server. A channel has five legal states, such as READY, IDLE, and so on. Each state represents a particular behavior in the connection between client and server at that moment in time. Clients may specify channel arguments to modify gRPC's default behavior, such as disabling message compression and so on.

J.4.2 Understanding request metadata

Metadata contains particular information about an RPC call, such as authentication details and so on. Metadata is provided in the form of a list of key-value pairs; keys are usually strings and the values can be of string or binary types. Though in most cases, values are provided as strings. Metadata helps the client to provide information about RPC messages to the server and vice versa. You can think of metadata as similar to headers in HTTP.

J.4.3 What is unary RPC?

Unary RPC represents a typical request-response pattern between client and server. gRPC supports this traditional model where requests and responses are exchanged in a sequential pattern. In this pattern, the client first calls the stub/client method, which invokes the particular method on the server. The server processes the messages and prepares and sends the response back to the client. In this model, the number of messages exchanged between client and server are equal (one response per request).

J.4.4 What is server streaming RPC?

In the server-streaming model, the server sends a stream of responses for a single client request. Server streaming can be used in scenarios where it makes sense to send multiple responses for a single client request. Imagine a scenario where you place an order on our retail store and the server starts processing the order by verifying the payment and completing the shipping request. The payment processing and shipping operations may be done in two parallel microservices on the server. Through server streaming, the server can now send an update to the client as soon as each step completes.

Once the server has sent all of its response messages to the client, it sends its status details (status code) and optional trailing metadata. The client uses this information to identify the end of the stream from the server.

J.4.5 What is client streaming RPC?

Similar to server streaming RPC, gRPC also supports client-streaming RPC. In this scenario, the client sends a stream of requests to the server to which the server typically (but not necessarily) sends back a single response. The server waits for the client to send its status details along with any optional trailing metadata before the server starts sending back the responses. Client streaming is useful in scenarios where the client needs to submit multiple inputs to the server over a period of time before the server can perform its

processing/calculations and provide the output. Imagine a scenario where you might take a metered taxi ride. The taxi (client) will upload its location data every few seconds or so. The server, upon receiving the location details, calculates the taxi fare based on the distance travelled and pushes an update to the client once every few minutes.

J.4.6 What is bidirectional streaming RPC?

In bidirectional streaming RPC, again the client initiates the call. The client application starts sending a stream of requests to the server and the server starts sending a stream of responses to the client. The order in which the data is exchanged is application dependent. The server may decide to wait until it has received all the client request messages before sending back the responses, or the server and client could send responses while the client is still sending request messages to the server.

K

Creating a certificate authority and related keys with OpenSSL

How do you build trust between a web site you visit (say, for example, Amazon) and the browser (a client application) you use to access it? A third party that's known to all the client applications (browsers) signs the certificates given to services such as Amazon. This third party is called a *certificate authority* (CA). Anyone who wants to expose services over the web that are protected with Transport Layer Security (TLS) must get their certificate signed by a trusted CA. Few trusted CAs are available globally, and their public keys are embedded in all browsers. When a browser talks to Amazon over TLS, it can verify that Amazon's certificate is valid (not forged) by verifying its signature against the corresponding CA's public key that's embedded in the browser. The certificate also includes the hostname of Amazon (which is the common name), so the browser knows it's communicating with the right server.

In this appendix, we show you how to create a certificate authority (CA) by using OpenSSL. OpenSSL is a commercial-grade toolkit and cryptographic library for TLS, available for multiple platforms. You can download and set up the distribution that fits your platform from <https://www.openssl.org/source>. But the easiest way to try OpenSSL is to use Docker. In this appendix, you use an OpenSSL Docker image. You need to install Docker, following the instructions at <https://docs.docker.com/install/#supported-platforms>. The process is straightforward. A deeper understanding of how Docker works isn't necessary to follow along in this appendix (we talk about Docker and containers in detail in appendix A).

K.1 Creating a certificate authority (CA)

Assuming that you have Docker installed, follow the instructions in this section to set up the CA. Let's spin up the OpenSSL Docker container from the appendix-k/sample01/ directory.

The following `docker run` command starts OpenSSL in a Docker container with a volume mount that maps the `appendix-k/sample01/` directory (or the current directory, which is indicated by `$(pwd)`) from the host file system to the `/export` directory of the container file system. This volume mount lets you share part of the host file system with the container file system. When the OpenSSL container generates certificates, those are written to the `/export` directory of the container file system. Because we have a volume mount, everything inside the `/export` directory of the container file system is also accessible from the `appendix-k/sample01/` directory of the host file system.

Listing K.1. Runs OpenSSL in a Docker container

```
\> docker run -it -v $(pwd):/export prabath/openssl
#
```

When you run the command in listing K.1 for the first time, it can take a couple of minutes to execute. It ends with a command prompt, where you can execute your OpenSSL commands to create the CA. Use the command in listing K.2 to generate a private key for the CA.

Listing K.2. Generates a private key for the certificate authority

```
# openssl genrsa -aes256 -passout pass:"manning123" -out /export/ca/ca_key.pem 4096
Generating RSA private key, 4096 bit long modulus
```

The generated key file is stored in the `/export/ca` directory of the Docker container (as specified by the `-out` argument). Then again, because you mapped the `appendix-k/sample01/` directory of the host file system to the `/export` directory, the generated key file is also available inside the `appendix-k/sample01/ca` directory of the host machine. In listing K.2, the `genrsa` command generates a private key of 4,096 bits and encrypts it with AES256 and the provided passphrase. We use `manning123` as the passphrase, which is passed to the `genrsa` command under the `-passout` argument.

NOTE The value of the passphrase must be prefixed with `pass:` and must be defined within double quotes.

Next, use the command (`req -new`) in listing K.3 to generate a public key that points to the already generated private key (`-key`) with an expiration time of 365 days (`-days`).

Listing K.3. Generates a public key for the certificate authority

```
# openssl req -new -passin pass:"manning123" -key /export/ca/ca_key.pem -x509 -days 365 -out
/export/ca/ca_cert.pem -subj "/CN=ca.ecomm.com"
```

While creating the public key, OpenSSL needs to know the details related to the organization behind the CA (country, state, organization name, and so on). Of those details, what matters most is the Common Name (CN). You need to provide something meaningful. CN may not be important here, but when a client application talks to a TLS-secured endpoint, the

client validates whether the hostname of the endpoint matches the value of CN in the certificate; if not, it rejects the certificate.

In listing K.3, we provide the value of CN under the `-subj` argument; make sure the corresponding value starts with a `/`. The `-out` argument specifies where to store the generated public key. Now you can find two files in the `appendix-k/sample01/ca` directory: `ca_cert.pem`, which is the public key, and `ca_key.pem`, which is the private key of the certificate authority.

K.2 Generating keys for an application

In this section, we discuss how to create a public/private key pair for an application and get those keys signed by the CA you created in the section K.1. This application can be a microservice, a web server, a client application, and so on. To generate the public/private key pair for an application, you're going to use the same OpenSSL Docker container started in section K.1. Run the command in listing K.4 to generate a private key for the application.

Listing K.4. Generates a private key for the application

```
# openssl genrsa -aes256 -passout pass:"manning123" -out /export/application/app_key.pem 4096
```

Listing K.4 generates the private key file (`app_key.pem`) inside the `appendix-k/sample01/application` directory. When you have the private key for the application, to get it signed by the CA, you need to create a certificate-signing request (CSR) first. Run the command in listing K.5 in OpenSSL Docker container command prompt. It produces a file named `csr-for-app`, which you have to share with your CA to get a signed certificate.

Listing K.5. Generates certificate signing request (CSR) for the application

```
# openssl req -passin pass:"manning123" -new -key /export/application/app_key.pem -out
/export/application/csr-for-app -subj "/CN=app.ecomm.com"
```

The OpenSSL command in listing K.6 gets the CSR generated in listing K.5, signed by the CA. The output of the following command is the signed certificate of the application (`app_cert.pem`). You can find it inside the `appendix-k/sample01/application` directory.

Listing K.6. Generates application's CA signed certificate

```
# openssl x509 -req -passin pass:"manning123" -days 365 -in /export/application/csr-for-app -
CA /export/ca/ca_cert.pem -CAkey /export/ca/ca_key.pem -set_serial 01 -out
/export/application/app_cert.pem
```

Now we have a private key and signed certificate for the application. For some Java applications (for example, Spring Boot microservices), we need to have this key stored in a Java keystore or a JKS. A JKS is a Java-specific key storage.

In listing K.7, we generate the JKS from the application's private key and the public certificate. In the first command of the listing, we remove the passphrase of the private key (`app_key.pem`), and in the second command, we create a single file (`application_keys.pem`) with both the private key and the public certificate. With the third command, we create a

keystore of type PKCS with these keys. At the end of the third command, you can find the PKCS keystore (`app.p12`) inside the `appendix-k/sample01/application` directory. Finally, in the last command, we use a Java Keytool to create a JKS file from the `app.p12` PKCS keystore. There we pass the passphrase of the source (`app.p12`) keystore under the argument `srcstorepass` and the passphrase of the destination (`app.jks`) keystore under the argument `deststorepass`. For `app.p12`, we used `manning123` as the keystore passphrase. For simplicity, we use the same passphrase for the destination keystore (`app.jks`) as well.

Listing K.7. Creates a Java keystore with application's private/public keys

```
# openssl rsa -passin pass:"manning123" -in /export/application/app_key.pem -out
   /export/application/app_key.pem

# cat /export/application/app_key.pem /export/application/app_cert.pem >>
   /export/application/application_keys.pem

# openssl pkcs12 -export -passout pass:"manning123" -in
   /export/application/application_keys.pem -out /export/application/app.p12

# keytool -importkeystore -srcstorepass Manning123 -srckeyst
ore /export/application/app.p12 -srcstoretype pkcs12 -deststorepass Manning123 -destkeystore
   /export/application/app.jks -deststoretype JKS
```