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.

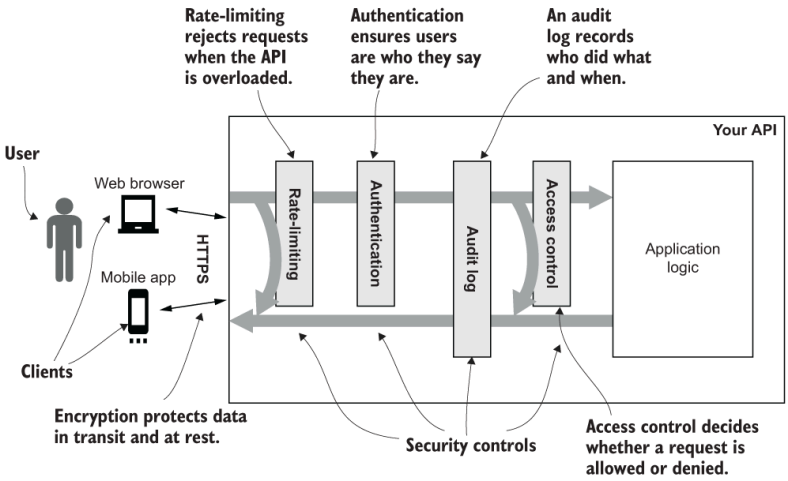

Figure 3.1 Applying security controls to the Natter API. Encryption prevents information disclosure. Rate-limiting protects availability. Authentication is used to ensure that users are who they say they are. Audit logging records who did what, to support accountability. Access control is then applied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

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 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 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.

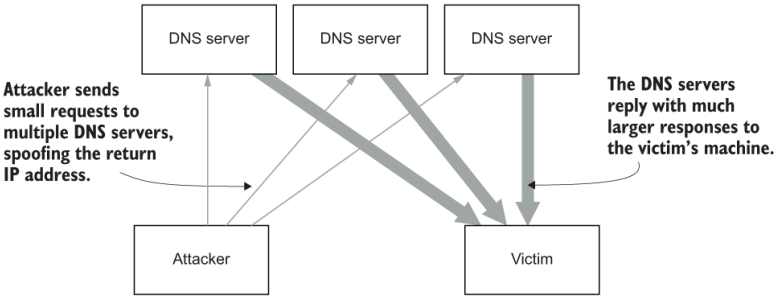

Figure 3.2 In a DNS amplification attack, the attacker sends the same DNS query to many DNS servers, spoofing their IP address to look like the request came from the victim. By carefully choosing the DNS query, the server can be tricked into replying with much more data than was in the original query,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

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 re-

jected 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.

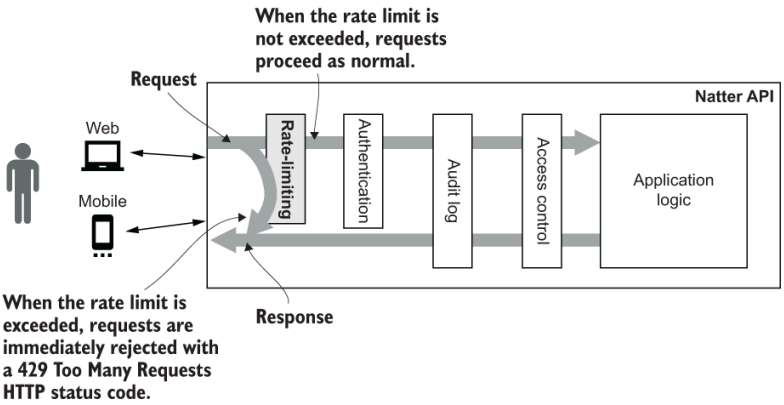

Figure 3.3 Rate-limiting rejects requests when your API is under too much load. By rejecting requests early before they have consumed too many resources, we can ensure that the requests we do process have enough resources to complete without errors. Rate-limiting should be the very first decision applied 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);          (1)

before((request, response) -> {
  if (!rateLimiter.tryAcquire()) {                   (2)
    response.header("Retry-After", "2");             (3)
    halt(429);                                       (4)
  }
});
```

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

(2) Check if the rate has been exceeded.

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

(4) Return a 429 Too Many Requests status.

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                                    (1)
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                                    (1)
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                                    (1)
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
HTTP/1.1 429 Too Many Requests                          (2)
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                          (2)
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
```

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

❷ 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?
    1. Rate-limiting should occur after access control.
    2. Rate-limiting stops all denial of service attacks.
    3. Rate-limiting should be enforced as early as possible.
    4. 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?
    1. Expires
    2. Retry-After
    3. Last-Modified
    4. Content-Security-Policy
    5. 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.

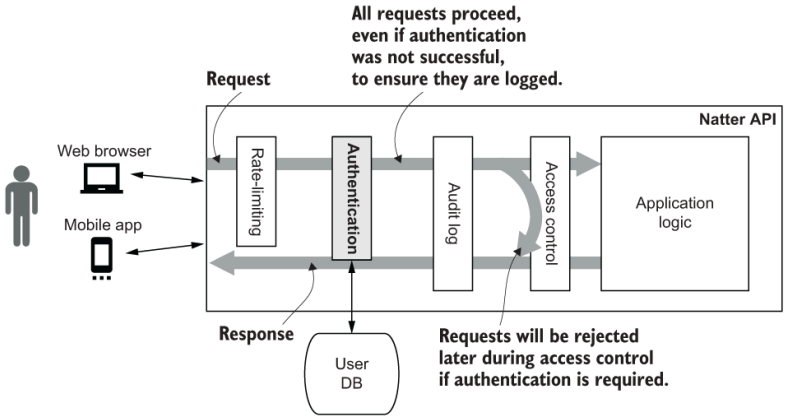

Figure 3.4 Authentication occurs after rate-limiting but before audit logging or access control. All requests proceed, even if authentication fails, to ensure that they are always logged. Unauthenticated requests will be rejected during access control, which occurs after audit logging.

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 colon4 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.

3.3.2 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 ex-

ample), 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());
    var username = json.getString("username");
    var password = json.getString("password");

    if (!username.matches(USERNAME_PATTERN)) {                    (1)
      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);      ❷
    database.updateUnique(                                    ❸
        "INSERT INTO users(user_id, pw_hash)" +
        " VALUES(?, ?)", username, hash);

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

❶ Apply the same username validation that you used before.

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

❸ 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");                    (1)
  if (authHeader == null || !authHeader.startsWith("Basic ")) {         (1)
    return;                                                             (1)
  }                                                                     (1)

  var offset = "Basic ".length();
  var credentials = new String(Base64.getDecoder().decode(              (2)
      authHeader.substring(offset)), StandardCharsets.UTF_8);           (2)

  var components = credentials.split(":", 2);                           (3)
  if (components.length != 2) {                                         (3)
    throw new IllegalArgumentException("invalid auth header");          (3)
  }                                                                     (3)

  var username = components[0];                                         (3)
  var password = components[1];                                         (3)

  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() &&                                               (4)
      SCryptUtil.check(password, hash.get())) {                         (4)
    request.attribute("subject", username);
  }
}
```

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

(2) Decode the credentials using Base64 and UTF-8.

(3) Split the credentials into username and password.

(4) 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.)

1. It should be easy to parallelize.
2. It should use a lot of storage on disk.
3. It should use a lot of network bandwidth.
4. It should use a lot of memory (several MB).
5. It should use a random salt for each password.
6. 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.)
    1. The password can be exposed in the `Referer` header.
    2. HTTPS slows down attackers trying to guess passwords.
    3. The password might be tampered with during transmission.
    4. Google penalizes websites in search rankings if they do not use HTTPS.
    5. 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.

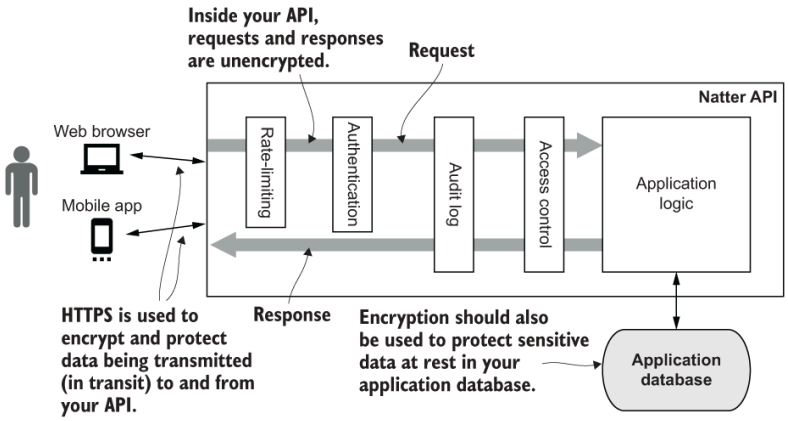

Figure 3.5 Encryption is used to protect data in transit between a client and our API, and at rest when stored in the database.

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."

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;                                    (1)

public class Main {
  public static void main(String... args) throws Exception {
    secure("localhost.p12", "changeit", null, null);                 (2)
    ..
  }
}
```

(1) Import the secure method.

❷ 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 `after-After` 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?
    1. Confidentiality
    2. Integrity

3. 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.

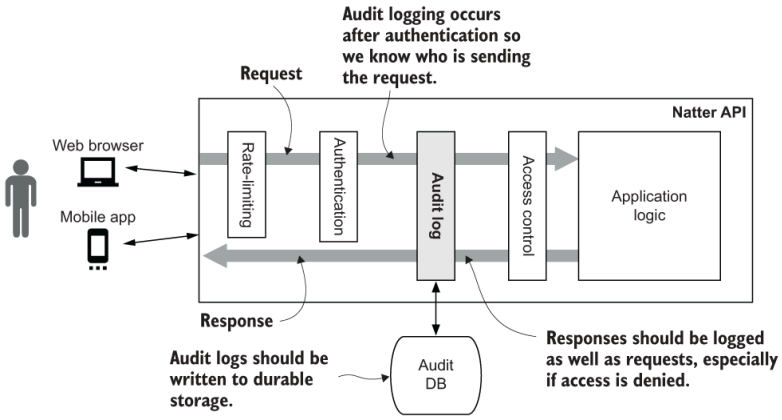

Figure 3.6 Audit logging should occur both before a request is processed and after it completes. When implemented as a filter, it should be placed after authentication, so that you know who is performing each action, but before access control checks so that you record operations that were attempted but 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(          (1)
          "SELECT NEXT VALUE FOR audit_id_seq");      (1)
      request.attribute("audit_id", auditId);         (1)
      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"),                (2)
        request.requestMethod(),
        request.pathInfo(),
        response.status(),
        request.attribute("subject"));
  }
}
```

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

(2) 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);          (1)
    var logs = database.findAll(AuditController::recordToJson,    (1)
            "SELECT * FROM audit_log " +                           (1)
                    "WHERE audit_time >= ? LIMIT 20", since);      (1)
    return new JSONArray(logs);                                    (2)
}

private static JSONObject recordToJson(ResultSet row)              (3)
```

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

❶ Read log entries for the last hour.

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

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

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

❶ 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?
   1. The Peter principle
   2. The principle of least privilege
   3. The principle of defense in depth
   4. The principle of separation of duties
   5. 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.

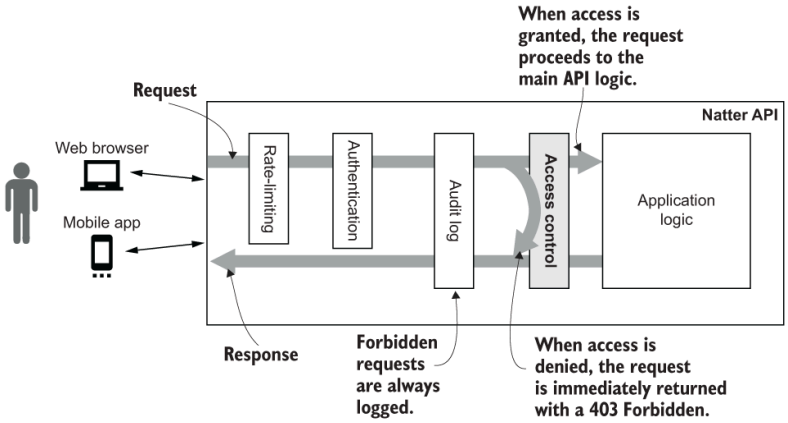

Figure 3.7 Access control occurs after authentication and the request has been logged for audit. If access is denied, then a forbidden response is immediately returned without running any of the application logic. If access is granted, then the request proceeds as normal.

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);                          (1)

before(auditController::auditRequestStart);                    (2)
afterAfter(auditController::auditRequestEnd);                  (2)
before("/spaces", userController::requireAuthentication);      (3)
post("/spaces", spaceController::createSpace);  ..
```

(1) First, try to authenticate the user.

(2) Then perform audit logging.

(3) 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(                                          (1)
          "INSERT INTO permissions(space_id, user_id, perms) " +      (1)
              "VALUES(?, ?, ?)", spaceId, owner, "rwd");              (1)

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

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

❶ 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) -> {                                    (1)
    if (!method.equalsIgnoreCase(request.requestMethod())) {         (2)
      return;                                                        (2)
    }                                                                (2)

    requireAuthentication(request, response);                        (3)

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

    var perms = database.findOptional(String.class,                  (4)
        "SELECT perms FROM permissions " +                           (4)
            "WHERE space_id = ? AND user_id = ?",                    (4)
        spaceId, username).orElse("");                               (4)

    if (!perms.contains(permission)) {                               (5)
      halt(403);                                                     (5)
    }                                                                (5)
  };
}
```

(1) Return a new Spark filter as a lambda expression.

(2) Ignore requests that don't match the request method.

(3) First check if the user is authenticated.

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

(5) 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);                              (1)

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

  before("/spaces",                                                  (2)
      userController::requireAuthentication);                        (2)
  post("/spaces",
```

```
        spaceController::createSpace);

    before("/spaces/:spaceId/messages",                    ❸
        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);          ❹

    ...
}
```

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

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

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

❹ 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?")) {                                    (1)
      throw new IllegalArgumentException("invalid permissions");
    }

    database.updateUnique(                                             (2)
        "INSERT INTO permissions(space_id, user_id, perms) " +
            "VALUES(?, ?, ?);", spaceId, userToAdd, perms);

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

(1) Ensure the permissions granted are valid.

(2) 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":"/spa
```

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)?
    1. 403 Forbidden
    2. 404 Not Found
    3. 401 Unauthorized
    4. 418 I'm a Teapot
    5. 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.

---

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.

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.

5. **https://tools.ietf.org/html/rfc5802**

6. **https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/**