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).

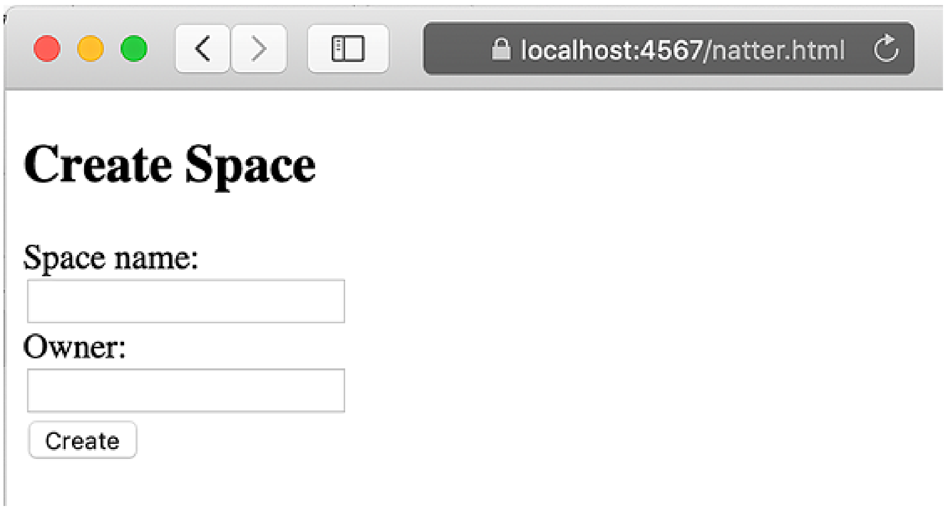


Figure 4.1 A simple web UI for creating a social space with the Natter API

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

❶ Use the Fetch API to call the Natter API endpoint.

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

❸ 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>      ❶
    <style type="text/css">
      input { margin-right: 100% }                                ❷
    </style>
  </head>
  <body>
    <h2>Create Space</h2>
    <form id="createSpace">                                       ❸
      <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>
```

❶ Include the natter.js script file.

❷ Style the form as you wish using CSS.

❸ 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) {                  ❶
    document.getElementById('createSpace')                     ❶
        .addEventListener('submit', processFormSubmit);        ❶
});
function processFormSubmit(e) {
    e.preventDefault();                                        ❷

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

    createSpace(spaceName, owner);                             ❸

    return false;
}
```

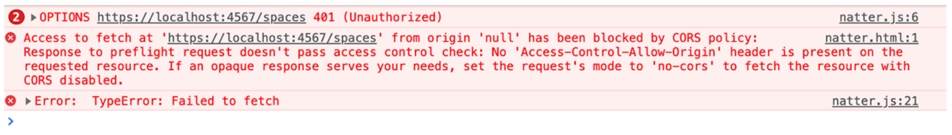
❶ When the document loads, add an event listener to intercept the form submission.

❷ Suppress the default form behavior.

❸ 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/.



```
2 ▸OPTIONS https://localhost:4567/spaces 401 (Unauthorized)                          natter.js:6
⊗ Access to fetch at 'https://localhost:4567/spaces' from origin 'null' has been blocked by CORS policy:   natter.html:1
  Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the
  requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with
  CORS disabled.
⊗ ▸Error:  TypeError: Failed to fetch                                                 natter.js:21
>
```

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.

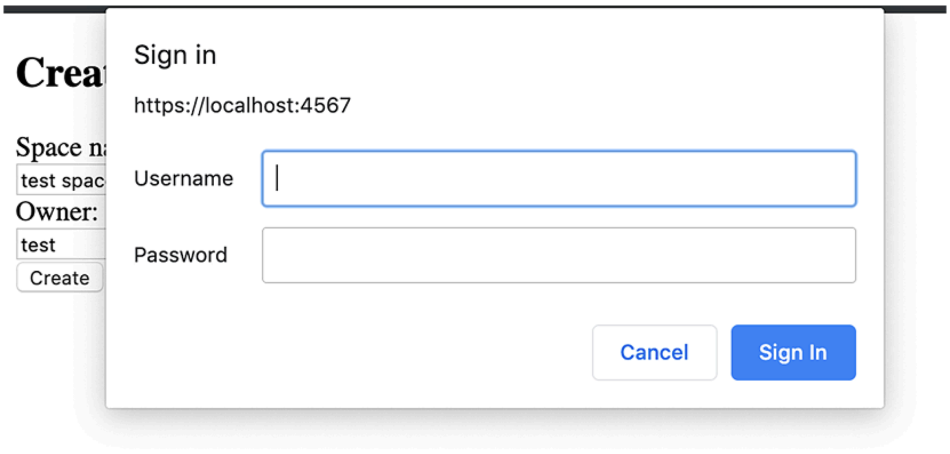

Sign in
https://localhost:4567

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 over-head. 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 experi-ence 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?
   1. http://api.example.com/test/1
   2. https:/ /api.example.com/test/2
   3. http://api.example.com:8443/test/2

4. https:/ /api.example.com:8443/test/2
5. 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 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.

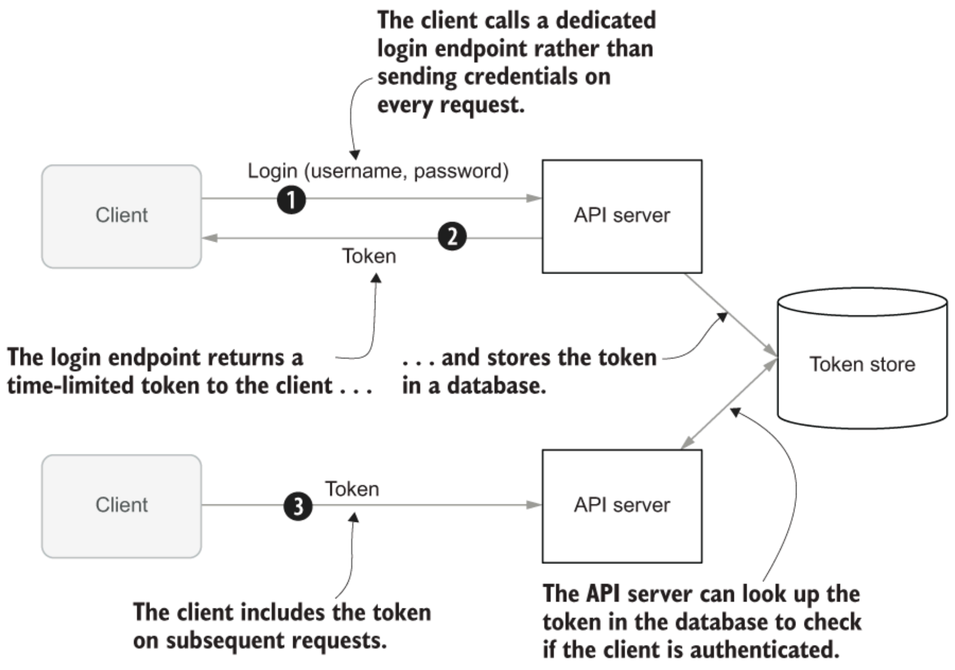

The client calls a dedicated login endpoint rather than sending credentials on every request.
Login (username, password)
1
Client
API server
2
Token
The login endpoint returns a time-limited token to the client . . .
. . . and stores the token in a database.
Token store
Client
3
Token
API server
The client includes the token on subsequent requests.
The API server can look up the token in the database to check if the client is authenticated.

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.

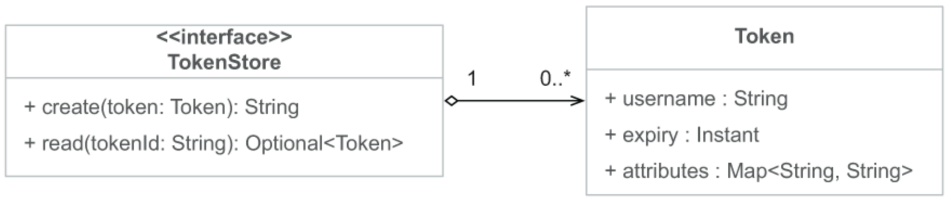
In the simplest case, this token store is a shared database indexed by the token ID, but more advanced (and loosely coupled) solutions are also possible, as you'll see in chapter 6. A short-lived token that is intended to authenticate a user while they are directly interacting with a site (or API) is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



<<interface>>
TokenStore
+ create(token: Token): String
+ read(tokenId: String): Optional<Token>
1
0..*
Token
+ username : String
+ expiry : Instant
+ attributes : Map<String, String>

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.

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);           (1)
  Optional<Token> read(Request request, String tokenId); (1)
```

```
    class Token {
      public final Instant expiry;                              ❷
      public final String username;                             ❷
      public final Map<String, String> attributes;              ❷
      public Token(Instant expiry, String username) {
        this.expiry = expiry;
        this.username = username;
        this.attributes = new ConcurrentHashMap<>();            ❸
      }
    }
  }
```

❶ A token can be created and then later looked up by token ID.

❷ A token has an expiry time, an associated username, and a set of attributes.

❸ 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.

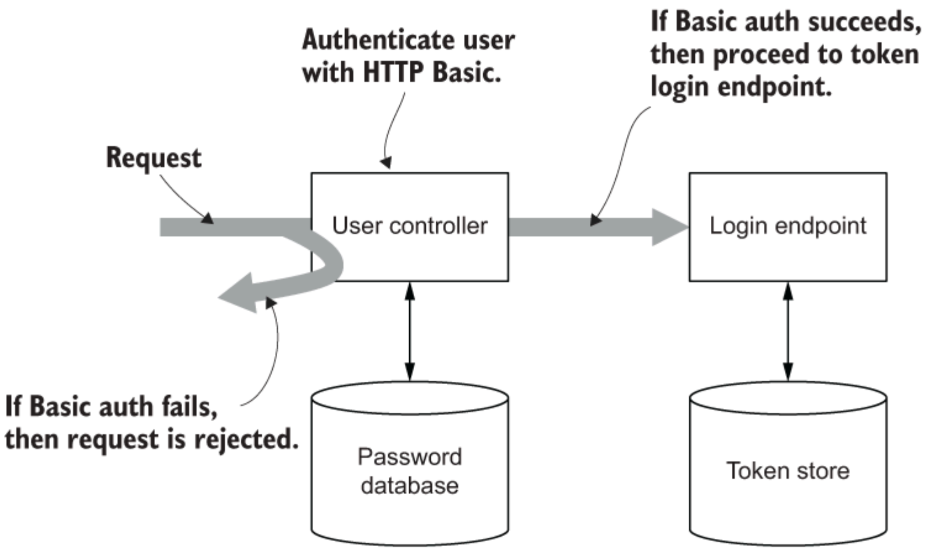

Authenticate user with HTTP Basic.
If Basic auth succeeds, then proceed to token login endpoint.
Request
User controller
Login endpoint
If Basic auth fails, then request is rejected.
Password database
Token store

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;                                1

    public TokenController(TokenStore tokenStore) {                     1
        this.tokenStore = tokenStore;                                   1
    }

    public JSONObject login(Request request, Response response) {
        String subject = request.attribute("subject");                  2
        var expiry = now().plus(10, ChronoUnit.MINUTES);                2

        var token = new TokenStore.Token(expiry, subject);              3
        var tokenId = tokenStore.create(request, token);                3

        response.status(201);
        return new JSONObject()                                         3
                .put("token", tokenId);                                 3
    }
}
```

1 Inject the token store as a constructor argument.

2 Extract the subject username from the request and pick a suitable expiry time.

3 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;                                           1
var tokenController = new TokenController(tokenStore);                  1
```

```
before(userController::authenticate);                          ❷
var auditController = new AuditController(database);           ❸
before(auditController::auditRequestStart);                    ❸
afterAfter(auditController::auditRequestEnd);                  ❸


before("/sessions", userController::requireAuthentication);    ❹
post("/sessions", tokenController::login);                     ❹
```

❶ Create the new TokenController, at first with a null TokenStore.

❷ Ensure the user is authenticated by the UserController first.

❸ Calls to the login endpoint should be logged, so make sure that also happens first.

❹ Reject unauthenticated requests before the login endpoint can be accessed.

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 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.

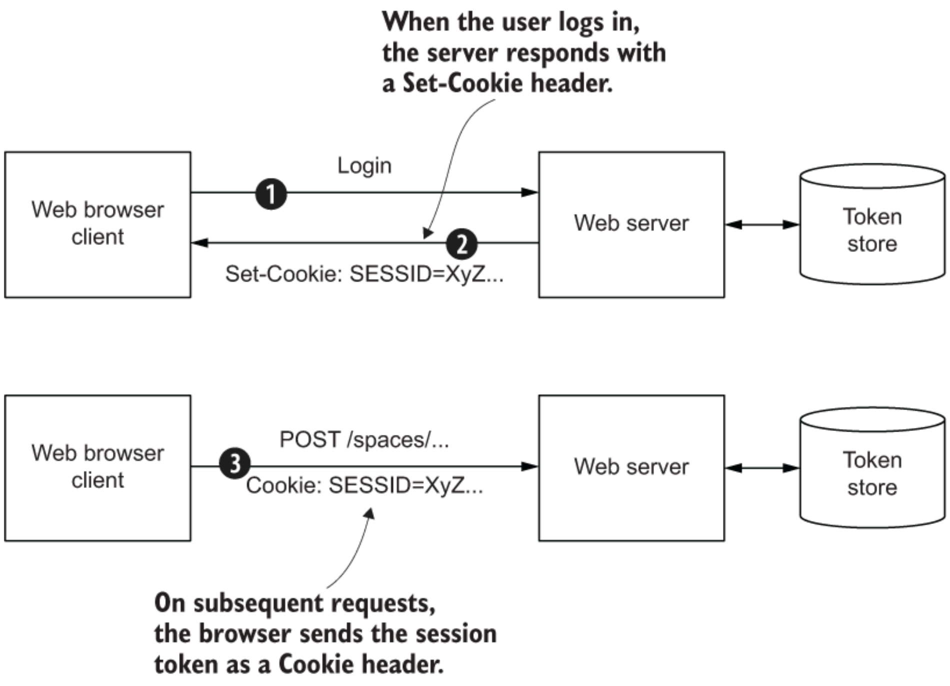

When the user logs in, the server responds with a Set-Cookie header.
Web browser client
Login
1
2
Set-Cookie: SESSID=XyZ...
Web server
Token store
Web browser client
3
POST /spaces/...
Cookie: SESSID=XyZ...
Web server
Token store
On subsequent requests, the browser sends the session token as a Cookie header.

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);                    1

        session.attribute("username", token.username);          2
        session.attribute("expiry", token.expiry);              2
        session.attribute("attrs", token.attributes);           2

        return session.id();
    }
    @Override
    public Optional<Token> read(Request request, String tokenId) {

        var session = request.session(false);                        ❸
        if (session == null) {
            return Optional.empty();
        }

        var token = new Token(session.attribute("expiry"),           ❹
                session.attribute("username"));                      ❹
        token.attributes.putAll(session.attribute("attrs"));         ❹

        return Optional.of(token);
    }
}
```

❶ Pass true to request.session() to create a new session cookie.

❷ Store token attributes as attributes of the session cookie.

❸ Pass false to request.session() to check if a valid session is present.

❹ 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 \                                           ❶
    -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                                                          ❷
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"}                           ❸
```

❶ Use the -u option to send HTTP Basic credentials.

❷ Spark returns a Set-Cookie header for the new session token.

❸ 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.

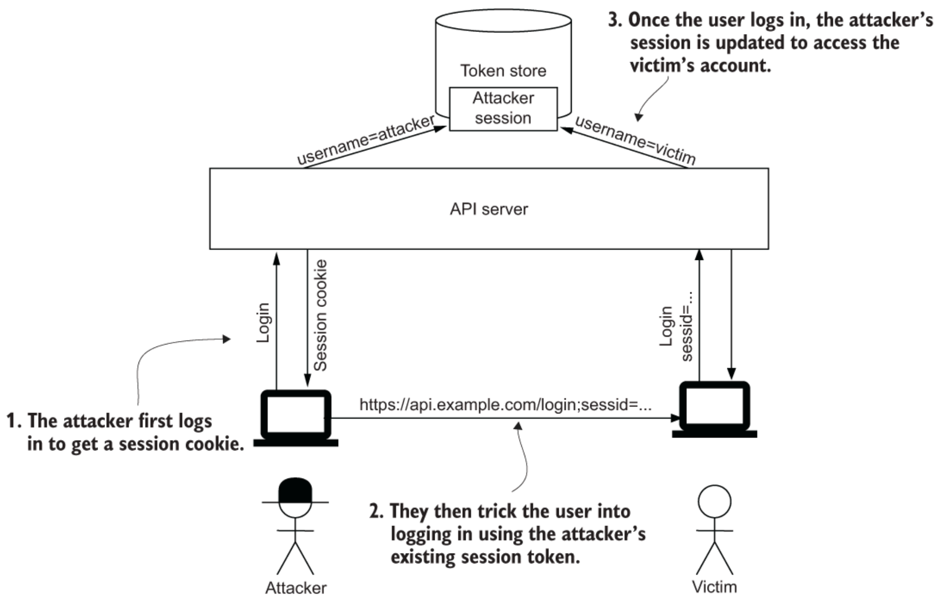

Token store
Attacker session
3. Once the user logs in, the attacker's session is updated to access the victim's account.
username=attacker
username=victim
API server
Login
Session cookie
Login
sessid=...
https://api.example.com/login;sessid=...
1. The attacker first logs in to get a session cookie.
2. They then trick the user into logging in using the attacker's existing session token.
Attacker
Victim

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.

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 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);    ①
        if (session != null) {                   ①
            session.invalidate();                ①
        }
        session = request.session(true);         ②
        session.attribute("username", token.username);
        session.attribute("expiry", token.expiry);
        session.attribute("attrs", token.attributes);
        return session.id();
    }
```

① Check if there is an existing session and invalidate it.

② 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.
`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 -> {        1
        if (now().isBefore(token.expiry)) {                    1
            request.attribute("subject", token.username);      2
            token.attributes.forEach(request::attribute);      2
        }
    });
}
```

❶ Check if a token is present and not expired.

❷ 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 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 \          ❶
  -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                         ❷
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":"node0l2q3fc024gw8wq4wp961y5rk0"}
```

❶ Use the -c option to save cookies from the response to a file.

❷ The server returns a Set-Cookie header for the session cookie.

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 \                        ❶
  -H 'Content-Type: application/json' \
  -d '{"name":"test space","owner":"test"}' \
  https://localhost:4567/spaces
{"name":"test space","uri":"/spaces/1"}         ❷
```

❶ Use the -b option to curl to send cookies from a cookie jar.

❷ The request succeeds as the session cookie was validated.

Pop quiz

2. What is the best way to avoid session fixation attacks?
   1. Ensure cookies have the Secure attribute.
   2. Only allow your API to be accessed over HTTPS.
   3. Ensure cookies are set with the HttpOnly attribute.
   4. Add a Content-Security-Policy header to the login response.
   5. Invalidate any existing session cookie after a user authenticates.
3. Which cookie attribute should be used to prevent session cookies being read from JavaScript?
   1. Secure
   2. HttpOnly
   3. Max-Age=-1
   4. SameSite=lax
   5. 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 the malicious site can create fake requests to your API that appear to come from a genuine client.

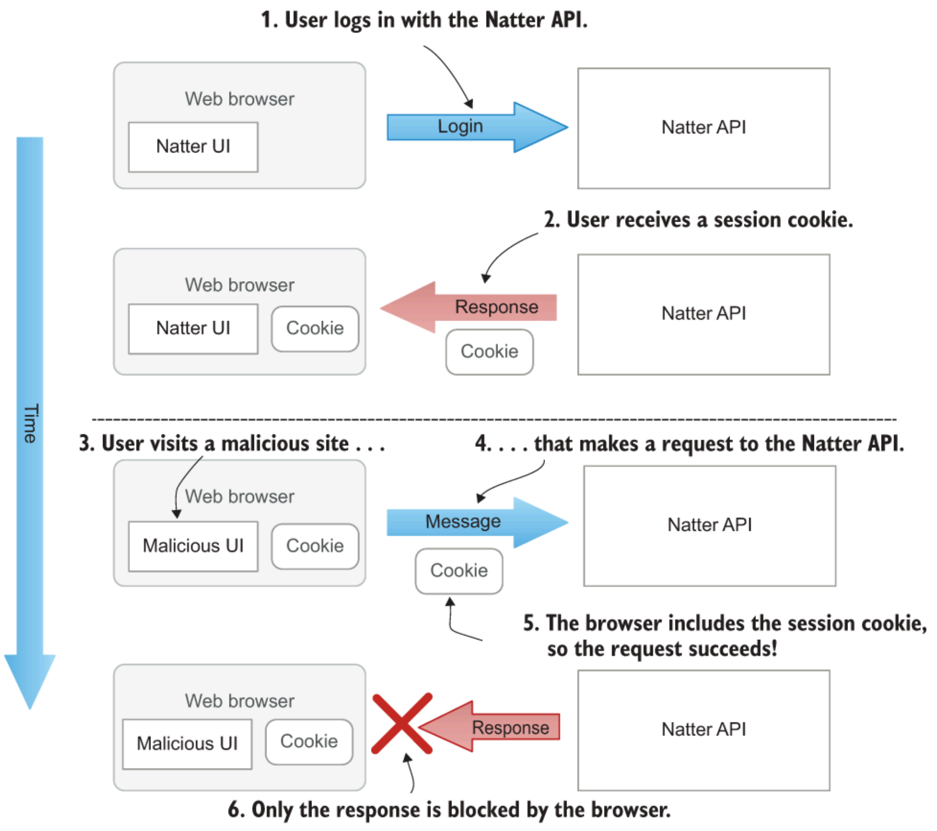

1. User logs in with the Natter API.
Web browser
Natter UI
Login
Natter API
2. User receives a session cookie.
Web browser
Natter UI
Cookie
Response
Cookie
Natter API
Time
3. User visits a malicious site . . .
4. . . . that makes a request to the Natter API.
Web browser
Malicious UI
Cookie
Message
Cookie
Natter API
5. The browser includes the session cookie, so the request succeeds!
Web browser
Malicious UI
Cookie
Response
Natter API
6. Only the response is blocked by the browser.

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.

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](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.

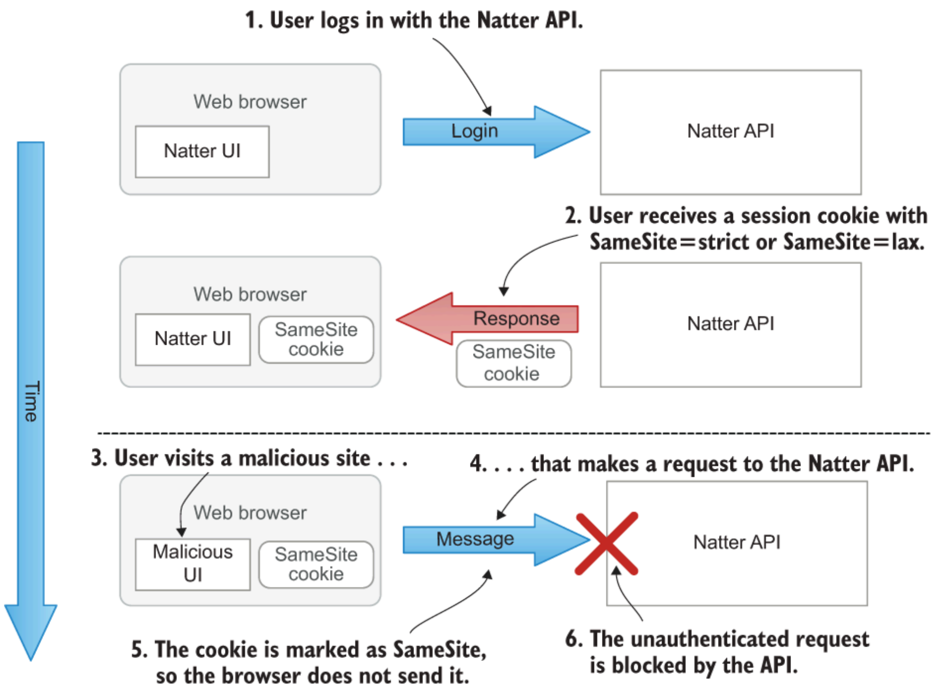

1. User logs in with the Natter API.
Web browser
Natter UI
Login
Natter API
2. User receives a session cookie with SameSite=strict or SameSite=lax.
Web browser
Natter UI
SameSite cookie
Response
SameSite cookie
Natter API
Time
3. User visits a malicious site . . .
4. . . . that makes a request to the Natter API.
Web browser
Malicious UI
SameSite cookie
Message
Natter API
5. The cookie is marked as SameSite, so the browser does not send it.
6. The unauthenticated request is blocked by the API.

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.

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.

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.

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.

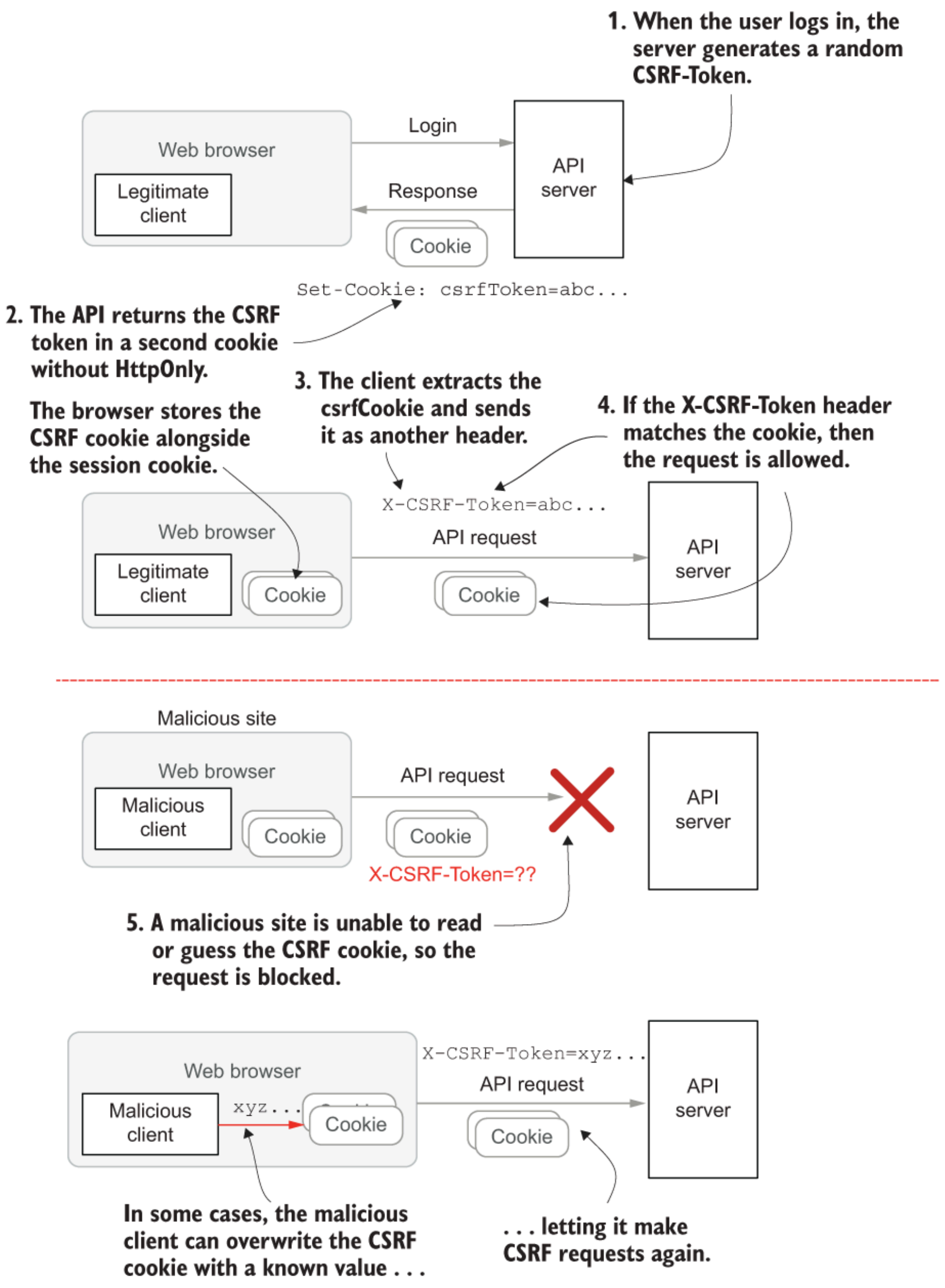
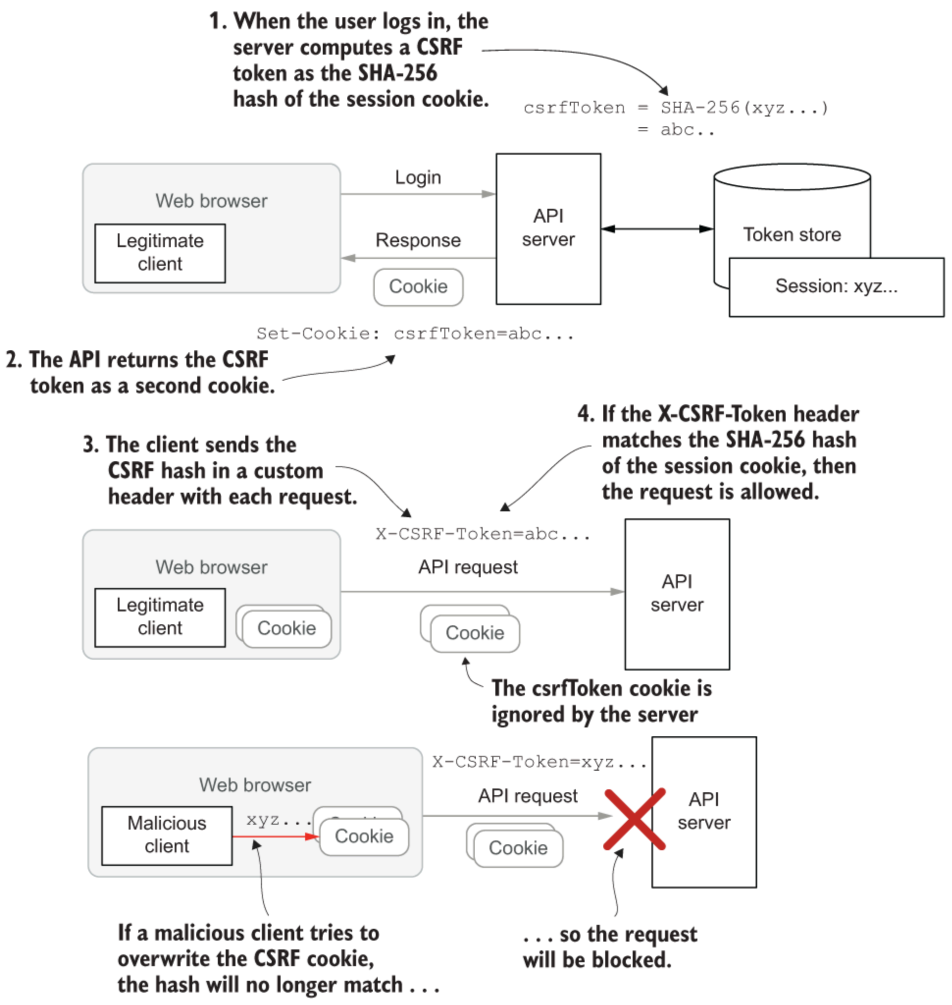

1. When the user logs in, the server generates a random CSRF-Token.
Web browser
Legitimate client
Login
Response
API server
Cookie
Set-Cookie: csrfToken=abc...
2. The API returns the CSRF token in a second cookie without HttpOnly.
The browser stores the CSRF cookie alongside the session cookie.
3. The client extracts the csrfCookie and sends it as another header.
4. If the X-CSRF-Token header matches the cookie, then the request is allowed.
X-CSRF-Token=abc...
Web browser
Legitimate client
Cookie
API request
Cookie
API server
Malicious site
Web browser
Malicious client
Cookie
API request
Cookie
X-CSRF-Token=??
API server
5. A malicious site is unable to read or guess the CSRF cookie, so the request is blocked.
Web browser
Malicious client
xyz...
Cookie
X-CSRF-Token=xyz...
API request
Cookie
API server
In some cases, the malicious client can overwrite the CSRF cookie with a known value . . .
. . . letting it make CSRF requests again.

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.

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.



1. When the user logs in, the server computes a CSRF token as the SHA-256 hash of the session cookie.
csrfToken = SHA-256(xyz...)
= abc..
Web browser
Legitimate client
Login
Response
Cookie
API server
Token store
Session: xyz...
Set-Cookie: csrfToken=abc...
2. The API returns the CSRF token as a second cookie.
3. The client sends the CSRF hash in a custom header with each request.
4. If the X-CSRF-Token header matches the SHA-256 hash of the session cookie, then the request is allowed.
X-CSRF-Token=abc...
API request
Web browser
Legitimate client
Cookie
Cookie
API server
The csrfToken cookie is ignored by the server
X-CSRF-Token=xyz...
API request
Web browser
Malicious client
xyz...
Cookie
Cookie
API server
If a malicious client tries to overwrite the CSRF cookie, the hash will no longer match . . .
. . . so the request will be blocked.

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.

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).

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 =         (1)
            Base64.getUrlEncoder().withoutPadding();      (1)
    private static final Base64.Decoder decoder =         (1)
            Base64.getUrlDecoder();                       (1)

    public static String encode(byte[] data) {            (2)
        return encoder.encodeToString(data);              (2)
    }                                                     (2)

    public static byte[] decode(String encoded) {         (2)
        return decoder.decode(encoded);                   (2)
    }                                                     (2)
}
```

(1) Define static instances of the encoder and decoder objects.

(2) 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 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 6410 (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()));           1
    }

    @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);                          ❷
        var computed = sha256(session.id());                               ❷

        if (!MessageDigest.isEqual(computed, provided)) {                  ❸
            return Optional.empty();                                       ❸
        }

        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");             ❹
            return sha256.digest(                                          ❹
                tokenId.getBytes(StandardCharsets.UTF_8));                 ❹
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException(e);
        }
    }
}
```

❶ Return the SHA-256 hash of the session cookie, Base64url-encoded.

❷ Decode the supplied token ID and compare it to the SHA-256 of the session.

❸ If the CSRF token doesn't match the session hash, then reject the request.

❹ 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");                ❶
    if (tokenId == null) return;                                  ❶

    tokenStore.read(request, tokenId).ifPresent(token -> {        ❷
        if (now().isBefore(token.expiry)) {
            request.attribute("subject", token.username);
            token.attributes.forEach(request::attribute);
        }
    });
}
```

❶ Read the CSRF token from the X-CSRF-Token header.

❷ 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;Ht
```

```
...
{"token":"gB7CiKkxx0FFsR4lhV9hsvA1nyT7Nw5YkJw_ysMm6ic"}
```

❶ 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.)
   1. http://www .example.com/test
   2. https:/ /other.com:8443/test
   3. https:/ /www .example.com:8443/test
   4. https:/ /www .example.org:8443/test
   5. 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?
   1. Insufficient crypto magic.
   2. Browsers may reject the second cookie.
   3. An attacker may be able to overwrite the second cookie.
   4. An attacker may be able to guess the second cookie value.
   5. 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.

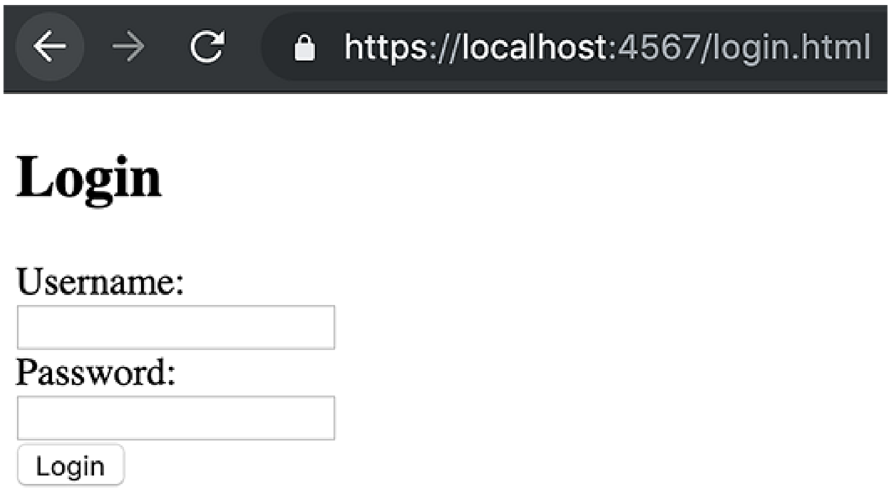


Figure 4.13 The login UI features a simple username and password form. Once successfully submitted, the form will redirect to the main natter.html UI page that you built earlier.

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 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);                                      1
    }
}
```

❶ 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% }                        ❶
        </style>
</head>
<body>
<h2>Login</h2>
<form id="login">
    <label>Username: <input name="username" type="text"         ❷
                            id="username">                      ❷
    </label>                                                    ❷
    <label>Password: <input name="password" type="password"     ❸
                            id="password">                      ❸
    </label>                                                    ❸
    <button type="submit">Login</button>
</form>
</body>
</html>
```

❶ As before, customize the CSS to style the form as you wish.

❷ The username field is a simple text field.

❸ 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);      ❶

    fetch(apiUrl + '/sessions', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
  'Authorization': credentials                                          ❶
        }
    })
    .then(res => {
       if (res.ok) {
         res.json().then(json => {
            document.cookie = 'csrfToken=' + json.token +              ❷
                ';Secure;SameSite=strict';                              ❷
            window.location.replace('/natter.html');                    ❷
         });
       }
    })
    .catch(error => console.error('Error logging in: ', error));        ❸
}

window.addEventListener('load', function(e) {                           ❹
    document.getElementById('login')                                    ❹
        .addEventListener('submit', processLoginSubmit);                ❹
});                                                                     ❹
```

```
function processLoginSubmit(e) {                                         4
    e.preventDefault();                                                  4

    let username = document.getElementById('username').value;           4
    let password = document.getElementById('password').value;           4

    login(username, password);                                           4
    return false;                                                        4
}
```

1 Encode the credentials for HTTP Basic authentication.

2 If successful, then set the csrfToken cookie and redirect to the Natter UI.

3 Otherwise, log the error to the console.

4 Set up an event listener to intercept form submit, just as you did for the Create Space UI.

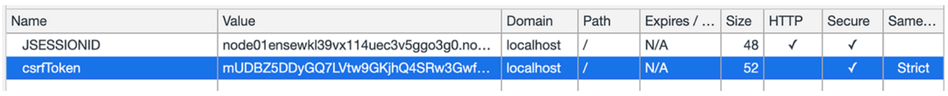
Rebuild and restart the API using

```
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(';')             ❶
        .map(item => item.split('=')                         ❷
            .map(x => decodeURIComponent(x.trim())))         ❸
        .filter(item => item[0] === cookieName)[0]           ❹

    if (cookieValue) {
        return cookieValue[1];
    }
}
```

❶ Split the cookie string into individual cookies.

❷ Then split each cookie into name and value parts.

❸ Decode each part.

❹ 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');                    ❶

    fetch(apiUrl + '/spaces', {
        method: 'POST',
        credentials: 'include',
        body: JSON.stringify(data),
        headers: {
            'Content-Type': 'application/json',
            'X-CSRF-Token': csrfToken                          ❷
        }
    })
    .then(response => {
        if (response.ok) {
            return response.json();
        } else if (response.status === 401) {                  ❸
            window.location.replace('/login.html');            ❸
        } else {
            throw Error(response.statusText);
        }
    })
    .then(json => console.log('Created space: ', json.name, json.uri))
    .catch(error => console.error('Error: ', error));
}
```

❶ Extract the CSRF token from the cookie.

❷ Include the CSRF token as the X-CSRF-Token header.

❸ 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);                  1
```

1 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 backend 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;

    var provided = Base64url.decode(tokenId);                1
    var computed = sha256(session.id());                     1

    if (!MessageDigest.isEqual(computed, provided)) {        1
        return;                                              1
    }

    session.invalidate();                                    2
}
```

1 Verify the anti-CSRF token as before.

2 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");                ❶
    if (tokenId == null)
        throw new IllegalArgumentException("missing token header");

    tokenStore.revoke(request, tokenId);                          ❷

    response.status(200);                                         ❸
    return new JSONObject();                                      ❸
}
```

❶ Get the token ID from the X-CSRF-Token header.

❷ Revoke the token.

❸ 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);     ❶
```

❶ 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.

---

[1] At the time of writing, this initiative has been paused due to the global COVID-19 pandemic.

[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.

[3] Removing a cookie can fail if the Path or Domain attributes do not exactly match, for example.