2 Secure API development

This chapter covers

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

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

2.1 The Natter API

You've had the perfect business idea. What the world needs is a new social network. You've got the name and the concept: Natter --the social network for coffee mornings, book groups, and other small gatherings. You've defined your minimum viable product, somehow received some funding, and now need to put together an API and a simple web client. You'll soon be the new Mark Zuckerberg, rich beyond your dreams, and considering a run for president.

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

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

2.1.1 Overview of the Natter API

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

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

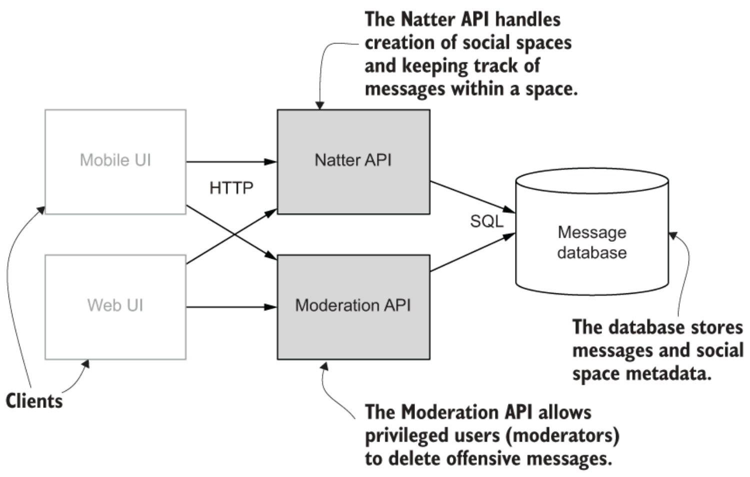
The Natter API handles creation of social spaces and keeping track of messages within a space.
Mobile UI
HTTP
Natter API
SQL
Message database
Web UI
Moderation API
The database stores messages and social space metadata.
Clients
The Moderation API allows privileged users (moderators) to delete offensive messages.

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

The Natter API offers the following operations:

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

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

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

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

2.1.2 Implementation overview

The Natter API is written in Java 11 using the Spark Java (**http://sparkjava.com**) framework (not to be confused with the Apache

Spark data analytics platform). To make the examples as clear as possible to non-Java developers, they are written in a simple style, avoiding too many Java-specific idioms. The code is also written for clarity and simplicity rather than production-readiness. Maven is used to build the code examples, and an H2 in-memory database (**https://h2database.com**) is used for data storage. The Dalesbred database abstraction library (**https://dalesbred.org**) is used to provide a more convenient interface to the database than Java's JDBC interface, without bringing in the complexity of a full object-relational mapping framework.

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

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

2.1.3 Setting up the project

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

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

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

```
natter-api
├── pom.xml                                        ❶
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── manning
    │               └── apisecurityinaction
    │                   └── App.java               ❷
    └── test
        └── java
            └── com
                └── manning
                    └── apisecurityinaction
                        └── AppTest.java           ❸
```

❶ The Maven project file

❷ The sample Java class generated by Maven

❸ A sample unit test file

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

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

Listing 2.1 pom.xml

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

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

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

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

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

❶ Configure Maven for Java 11.

❷ Set the main class for running the sample code.

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

❹ Include slf4j to enable debug logging for Spark.

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

2.1.4 Initializing the database

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

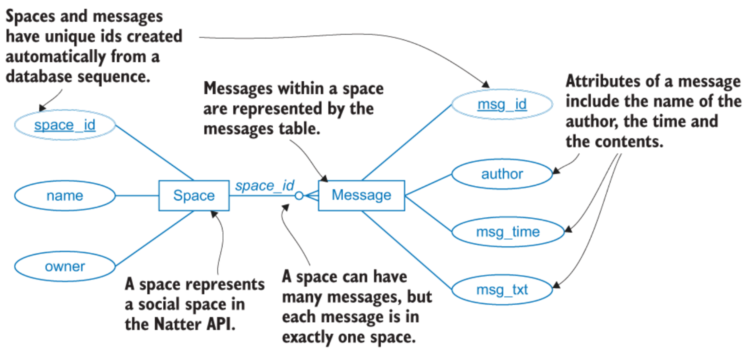
Spaces and messages have unique ids created automatically from a database sequence.
Messages within a space are represented by the messages table.
Attributes of a message include the name of the author, the time and the contents.
space_id
name
owner
Space
space_id
Message
msg_id
author
msg_time
msg_txt
A space represents a social space in the Natter API.
A space can have many messages, but each message is in exactly one space.

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

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

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

Listing 2.2 The database schema: schema.sql

```
CREATE TABLE spaces(                                   ❶
    space_id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    owner VARCHAR(30) NOT NULL
);
```

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

❶ The spaces table describes who owns which social spaces.

❷ We use sequences to ensure uniqueness of primary keys.

❸ The messages table contains the actual messages.

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

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

Listing 2.3 Setting up the database connection pool

```
package com.manning.apisecurityinaction;

import java.nio.file.*;

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

public class Main {

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

  private static void createTables(Database database)
      throws Exception {
    var path = Paths.get(                                       ❷
        Main.class.getResource("/schema.sql").toURI());         ❷
    database.update(Files.readString(path));                    ❷
  }
}
```

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

❷ Load table definitions from schema.sql.

2.2 Developing the REST API

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

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

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

2.2.1 Creating a new space

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

The code follows the general pattern of many API operations.

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

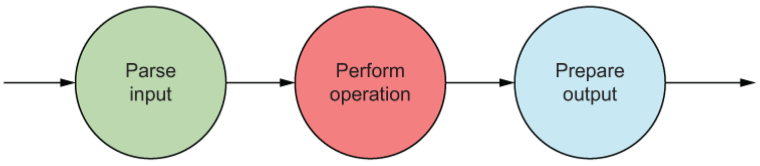

Figure 2.3 An API operation can generally be separated into three phases: first we parse the input and extract variables of interest, then we perform the actual operation, and finally we prepare some output that indicates the status of the operation.

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

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

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

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

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

Listing 2.4 Creating a new social space

```
package com.manning.apisecurityinaction.controller;
import org.dalesbred.Database;
import org.json.*;
import spark.*;
public class SpaceController {
  private final Database database;
  public SpaceController(Database database) {
    this.database = database;
  }
  public JSONObject createSpace(Request request, Response response)
      throws SQLException {
    var json = new JSONObject(request.body());                   ❶
    var spaceName = json.getString("name");
    var owner = json.getString("owner");
    return database.withTransaction(tx -> {                      ❷
      var spaceId = database.findUniqueLong(                     ❸
          "SELECT NEXT VALUE FOR space_id_seq;");                ❸
      // WARNING: this next line of code contains a
      // security vulnerability!
      database.updateUnique(
          "INSERT INTO spaces(space_id, name, owner) " +
              "VALUES(" + spaceId + ", '" + spaceName +
              "', '" + owner + "');");
```

```
            response.status(201);                                  ④
            response.header("Location", "/spaces/" + spaceId);     ④
            return new JSONObject()
                .put("name", spaceName)
                .put("uri", "/spaces/" + spaceId);
          });
        }
      }
```

① Parse the request payload and extract details from the JSON.

② Start a database transaction.

③ Generate a fresh ID for the social space.

④ Return a 201 Created status code with the URI of the space in the Location header.

2.3 Wiring up the REST endpoints

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

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

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

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

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

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

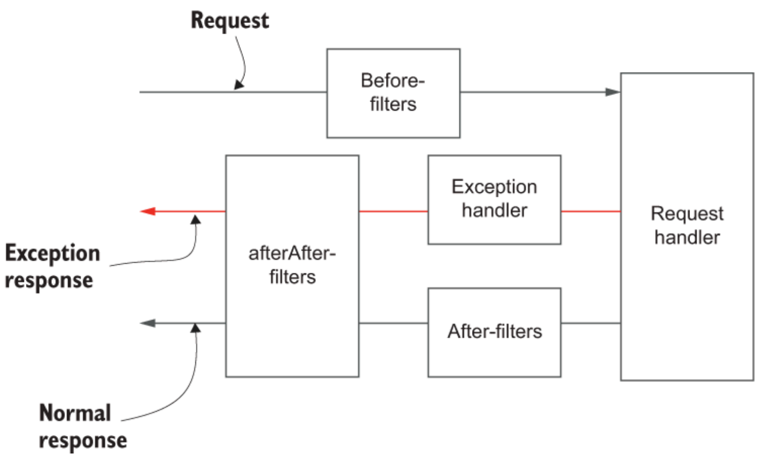

Figure 2.4 Spark before-filters run before the request is processed by your request handler. If the handler completes normally, then Spark will run any after-filters. If the handler throws an exception, then Spark runs the matching exception handler instead of the after-filters. Finally, afterAfter-filters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

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

Listing 2.5 The Natter REST API endpoints

```
package com.manning.apisecurityinaction;

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

import java.nio.file.*;

import static spark.Spark.*;                    (1)

public class Main {

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

    var spaceController =
        new SpaceController(database);          (2)
    post("/spaces",                             (3)
        spaceController::createSpace);          (3)

    after((request, response) -> {              (4)
      response.type("application/json");        (4)
    });

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

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

❶ Use static imports to use the Spark API.

❷ Construct the SpaceController and pass it the Database object.

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

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

2.3.1 Trying it out

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

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

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

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

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

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

2.4 Injection attacks

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

The OWASP Top 10

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

Web application top 10	API security top 10
A1:2017 - Injection	API1:2019 - Broken Object Level Authorization
A2:2017 - Broken Authentication	API2:2019 - Broken User Authentication
A3:2017 - Sensitive Data Exposure	API3:2019 - Excessive Data Exposure
A4:2017 - XML External Entities (XXE)	API4:2019 - Lack of Resources & Rate Limiting
A5:2017 - Broken Access Control	API5:2019 - Broken Function Level Authorization
A6:2017 - Security Misconfiguration	API6:2019 - Mass Assignment
A7:2017 - Cross-Site Scripting (XSS)	API7:2019 - Security Misconfiguration
A8:2017 - Insecure Deserialization	API8:2019 - Injection
A9:2017 - Using Components with Known Vulnerabilities	API9:2019 - Improper Assets Management
A10:2017 - Insufficient Logging & Monitoring	API10:2019 - Insufficient Logging & Monitoring

It's important to note that although every vulnerability in the Top 10 is worth learning about, avoiding the Top 10 will not by itself make your application secure. There is no simple checklist of vulnerabilities to avoid. Instead, this book will teach you the general principles to avoid entire classes of vulnerabilities.
An injection attack can occur anywhere that you execute dynamic code in response to user input, such as SQL and LDAP queries, and when running operating system commands.

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

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

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

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

Header and log injection

There are examples of injection vulnerabilities that do not involve code being executed at all. For example, HTTP headers are lines of text separated by carriage return and new line characters ("`\r\n`" in Java). If you include unvalidated user input in a HTTP header then an attacker may be able to add a "`\r\n`" character sequence and then inject their own HTTP headers into the response. The same can happen when you include user-

controlled data in debug or audit log messages (see chapter 3), allowing an attacker to inject fake log messages into the log file to confuse somebody later attempting to investigate an attack.
The Natter `createSpace` operation is vulnerable to a SQL injection attack because it constructs the command to create the new social space by concatenating user input directly into a string. The result is then sent to the database where it will be interpreted as a SQL command. Because the syntax of the SQL command is a string and the user input is a string, the database has no way to tell the difference.

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

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

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

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

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

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

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

```
org.h2.jdbc.JdbcSQLException: Syntax error in SQL statement "INSERT INTO spaces(space_id, n
```

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

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

```
Transfer-Encoding: chunked
Server: Jetty(9.4.8.v20171121)

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

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

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

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

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

Oh dear. It seems that by passing in carefully crafted input your user has managed to delete the spaces table entirely, and your whole social network with it! Figure 2.5 shows what the database saw when you executed the first curl command with the funny owner name. Because the user input values are concatenated into the SQL as strings, the database ends up seeing a single string that appears to contain two different statements: the `INSERT` statement we intended, and a `DROP` `TABLE` statement that the attacker has managed to inject. The first character of the owner name is a single quote character, which closes the open quote inserted by our code. The next two characters are a close parenthesis and a semicolon, which together ensure that the `INSERT` statement is properly terminated. The `DROP` `TABLE` statement is then inserted (injected) after the `INSERT` statement. Finally, the attacker adds another semicolon and two hyphen characters, which starts a comment in SQL. This ensures that the final close quote and parenthesis inserted by the code are ignored by the database and do not cause a syntax error.

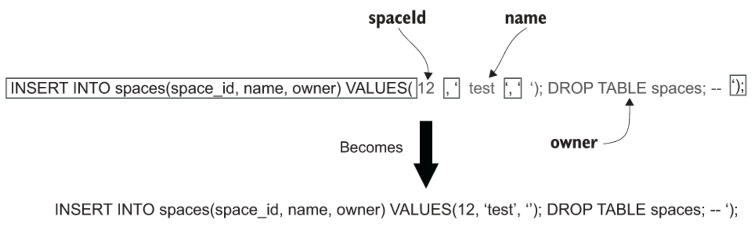
spaceId
name
INSERT INTO spaces(space_id, name, owner) VALUES(
12
,'
test
','
'); DROP TABLE spaces; --
');
Becomes
owner
INSERT INTO spaces(space_id, name, owner) VALUES(12, 'test', ''); DROP TABLE spaces; -- ');

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

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

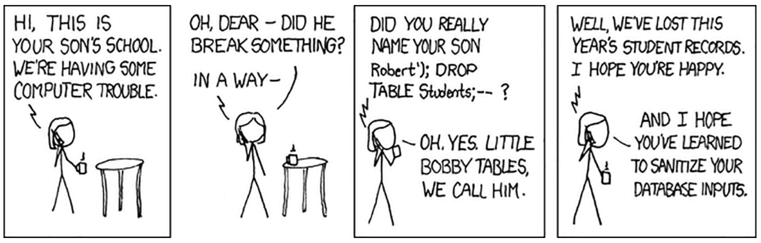
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

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

2.4.1 Preventing injection attacks

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

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

The best approach is to ensure that user input is always clearly separated from dynamic code by using APIs that support prepared statements. A prepared statement allows you to write the command or query that you want to execute with placeholders in it for user input, as shown in figure 2.7. You then separately pass the user input values and the database API ensures they are never treated as statements to be executed.

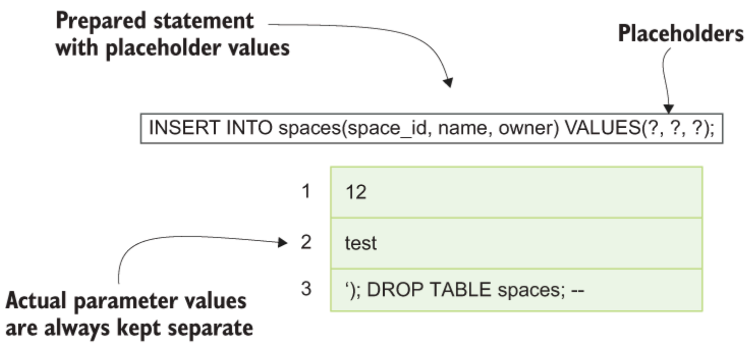
Prepared statement with placeholder values
Placeholders
INSERT INTO spaces(space_id, name, owner) VALUES(?, ?, ?);
1
12
2
test
3
‘); DROP TABLE spaces; --
Actual parameter values are always kept separate

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

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

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

Listing 2.6 Using prepared statements

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

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

      database.updateUnique(
          "INSERT INTO spaces(space_id, name, owner) " +          (1)
              "VALUES(?, ?, ?);", spaceId, spaceName, owner);     (1)

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

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

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

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

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

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

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

2.4.2 Mitigating SQL injection with permissions

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

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

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

Listing 2.7 Creating a restricted database user

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

❶ Create the new database user.

❷ Grant just the permissions it needs.

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

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

❶ Initialize the database schema as the privileged user.

❷ Switch to the natter_ api_user and recreate the database objects.

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

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

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

Pop quiz

1. Which one of the following is not in the 2017 OWASP Top 10?
   1. Injection
   2. Broken Access Control
   3. Security Misconfiguration
   4. Cross-Site Scripting (XSS)
   5. Cross-Site Request Forgery (CSRF)
   6. Using Components with Known Vulnerabilities
2. Given the following insecure SQL query string:

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

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

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

   what will be the output of running the query (assuming that the `users` table exists with a `password` column)?
   1. Nothing
   2. A syntax error
   3. John’s password
   4. The passwords of all users
   5. An integrity constraint error
   6. The messages written by John
   7. Any messages written by John and the passwords of all users

The answers are at the end of the chapter.

2.5 Input validation

Security flaws often occur when an attacker can submit inputs that violate your assumptions about how the code should operate. For example, you might assume that an input can never be more than a certain size. If you're using a language like C or C++ that lacks memory safety, then failing to check this assumption can lead to a serious class of attacks known as buffer overflow attacks. Even in a memory-safe language, failing to check that the inputs to an API match the developer's assumptions can result in unwanted behavior.

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

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

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

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

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

Insecure deserialization

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

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

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

See **https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/** for a description of the vulnerability. The vulnerability relies on a feature of Jackson that is disabled by default.
Although the API is using a safe JSON parser, it's still trusting the input in other regards. For example, it doesn't check whether the supplied username is less than the 30-character maximum configured in the database schema. What happens you pass in a longer username?

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

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

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

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

But you shouldn't rely on the database to catch all errors. A database is a valuable asset that your API should be protecting from invalid requests. Sending requests to the database that contain basic errors just ties up resources that you would rather use processing genuine requests. Furthermore, there may be additional constraints that are harder to express in a database schema. For example, you might require that the user exists in the corporate LDAP directory. In listing 2.8, you'll add some basic input validation to ensure that usernames are at most 30 characters long, and space names up to 255 characters. You'll also ensure that usernames contain only alphanumeric characters, using a regular expression.

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

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

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

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

Listing 2.8 Validating inputs

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

❶ Check that the space name is not too long.

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

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

ReDoS Attacks

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

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

Java 11 appears to be less susceptible to these attacks than earlier versions.
If you compile and run this new version of the API, you'll find that you still get a 500 error, but at least you are not sending invalid requests to the database anymore. To communicate a more descriptive error back to the user, you can install a Spark exception handler in your `Main` class, as shown in listing 2.9. Go back to the Main.java file in your editor and navigate to the end of the main method. Spark exception handlers are registered by calling the `Spark.exception()` method, which we have already statically imported. The method takes two arguments: the exception class to handle, and then a handler function that will take the exception, the request, and the response objects. The handler function can then use the response object to produce an appropriate error message. In this case, you will catch `IllegalArgumentException` thrown by our validation code, and `JSONException` thrown by the JSON parser when given incorrect input. In both cases, you can use a helper method to return a formatted 400 Bad Request error to the user. You can also return a 404 Not Found result when a user tries to access a space that doesn't exist by catching Dalesbred's `EmptyResultException`.

Listing 2.9 Handling exceptions

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

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

❶ Add required imports.

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

❸ Also handle exceptions from the JSON parser.

❹ Return 404 Not Found for Dalesbred empty result exceptions.

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

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

{"error": "java.lang.IllegalArgumentException: invalid username: a really long username tha
```

Pop quiz

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

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

    and recalling from the start of section 2.5 that Java is a memory-safe language, what is the main vulnerability an attacker could exploit in this code?
    1. Passing a negative message length
    2. Passing a very large message length
    3. Passing an invalid value for the message length
    4. Passing a message length that is longer than the buffer size
    5. Passing a message length that is shorter than the buffer size

The answer is at the end of the chapter.

2.6 Producing safe output

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

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

{"error": "java.lang.IllegalArgumentException: invalid username: a really long username tha
```

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

1. It includes details of the exact Java exception that was thrown. Although not a vulnerability by itself, these kinds of details in outputs help a potential attacker to learn what technologies are being used to power an API. The headers are also leaking the version of the Jetty webserver that is being used by Spark under the hood. With these details the attacker can try and find known vulnerabilities to exploit. Of course, if there are vulnerabilities then they may find them anyway, but you've made their job a lot easier by giving away these details. Default error pages often leak not just class names, but full stack traces and other debugging information.

2. It echoes back the erroneous input that the user supplied in the response and doesn't do a good job of escaping it. When the API client might be a web browser, this can result in a vulnerability known as reflected cross-site scripting (XSS). You'll see how an attacker can exploit this in section 2.6.1.
3. The Content-Type header in the response is set to `text/html` rather than the expected `application/json`. Combined with the previous issue, this increases the chance that an XSS attack could be pulled off against a web browser client.

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

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

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

```
  private static void badRequest(Exception ex,
      Request request, Response response) {
    response.status(400);
    response.body("{\"error\": \"" + ex.getMessage() + "\"}");
  }
```

Cross-Site Scripting

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

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

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

2.6.1 Exploiting XSS Attacks

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

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

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

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

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

Listing 2.10 Exploiting a reflected XSS

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

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

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

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

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

1. When the form is submitted, the browser sends a POST request to http://localhost:4567/spaces with a Content-Type header of text/plain and the hidden form field as the value. When the browser submits the form, it takes each form element and submits them as name=value pairs. The `&lt;`, `&gt;` and `&apos;` HTML entities are replaced with the literal values <, >, and `'` respectively.
2. The name of your hidden input field is `'{"x":"'`, although the value is your long malicious script. When the two are put together the API will see the following form input:

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

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

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

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

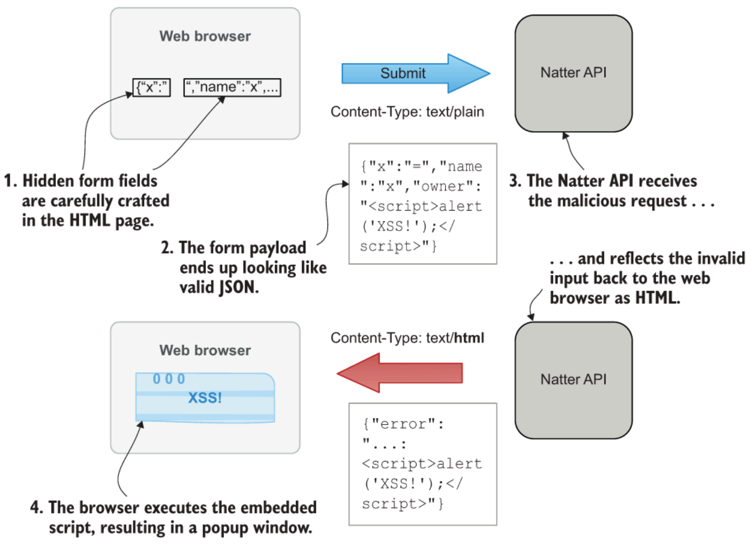
Web browser
{"x":"
","name":"x",...
Submit
Content-Type: text/plain
Natter API
1. Hidden form fields are carefully crafted in the HTML page.
{"x":"=","name
":"x","owner":
"<script>alert
('XSS!');</
script>"}
2. The form payload ends up looking like valid JSON.
3. The Natter API receives the malicious request . . .
. . . and reflects the invalid input back to the web browser as HTML.
Content-Type: text/html
Web browser
XSS!
Natter API
{"error":
"...:
<script>alert
('XSS!');</
script>"}
4. The browser executes the embedded script, resulting in a popup window.

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

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

2.6.2 Preventing XSS

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

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

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

Table 2.1 Useful security headers

Security header	Description	Comments
`X-XSS-Protection`	Tells the browser whether to block/ignore suspected XSS attacks.	The current guidance is to set to "`0`" on API responses to completely disable these protections due to security issues they can introduce.
`X-Content-Type-Options`	Set to `nosniff` to prevent the browser guessing the correct Content-Type.	Without this header, the browser may ignore your `Content-Type` header and guess (sniff) what the content really is. This can cause JSON output to be interpreted as HTML or JavaScript, so always add this header.
`X-Frame-Options`	Set to `DENY` to prevent your API responses being loaded in a frame or iframe.	In an attack known as drag 'n' drop clickjacking, the attacker loads a JSON response into a hidden iframe and tricks a user into dragging the data into a frame controlled by the attacker, potentially revealing sensitive information. This header prevents this attack in older browsers but has been replaced by Content Security Policy in newer browsers (see below). It is worth setting both headers for now.
`Cache-Control` and `Expires`	Controls whether browsers and proxies can cache content in the response and for how long.	These headers should always be set correctly to avoid sensitive data being retained in the browser or network caches. It can be useful to set default cache headers in a `before()` filter, to allow specific endpoints to override it if they have more specific caching requirements. The safest default is to disable caching completely using the `no-store` directive and then selectively re-enable caching for individual requests if necessary. The `Pragma: no-cache` header can be used to disable caching for older HTTP/1.0 caches.

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

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

Recommended CSP directives for REST responses

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

2.6.3 Implementing the protections

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

First, add a `before ()` filter that runs before each request and checks that any POST body submitted to the API has a correct Content-Type header of `application/ json`. The Natter API only accepts input from POST requests, but if your API handles other request methods that may contain a body (such as PUT or PATCH requests), then you should also enforce this filter for those methods. If the content type is incorrect, then you should return a 415 Unsupported Media Type status, because this is the standard status code for this case. You should also explicitly indicate the UTF-8 character-encoding in the response, to avoid tricks for stealing JSON data by specifying a different encoding such as UTF-16BE (see **https://portswigger.net/blog/json-hijacking-for-the-modern-web** for details).

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

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

Listing 2.11 Hardening your REST endpoints

```
public static void main(String... args) throws Exception {
  ..
  before(((request, response) -> {
    if (request.requestMethod().equals("POST") &&                 1
        !"application/json".equals(request.contentType())) {      1
      halt(415, new JSONObject().put(                             2
          "error", "Only application/json supported"
      ).toString());
    }
  }));

  afterAfter((request, response) -> {                             3
    response.type("application/json;charset=utf-8");
    response.header("X-Content-Type-Options", "nosniff");
    response.header("X-Frame-Options", "DENY");
    response.header("X-XSS-Protection", "0");
    response.header("Cache-Control", "no-store");
    response.header("Content-Security-Policy",
        "default-src 'none'; frame-ancestors 'none'; sandbox");
    response.header("Server", "");
  });

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

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

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

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

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

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

❹ Use a proper JSON library for all outputs.

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

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

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

Pop quiz

4. Which security header should be used to prevent web browsers from ignoring the `Content-Type` header on a response?
   1. `Cache-Control`
   2. `Content-Security-Policy`
   3. `X-Frame-Options: deny`
   4. `X-Content-Type-Options: nosniff`
   5. `X-XSS-Protection: 1; mode=block`
5. Suppose that your API can produce output in either JSON or XML format, according to the `Accept` header sent by the client. Which of the following should you not do? (There may be more than one correct answer.)
   1. Set the `X-Content-Type-Options` header.
   2. Include un-sanitized input values in error messages.

3. Produce output using a well-tested JSON or XML library.
4. Ensure the Content-Type is correct on any default error responses.
5. Copy the `Accept` header directly to the `Content-Type` header in the response.

The answers are at the end of the chapter.

Answers to pop quiz questions

1. e. Cross-Site Request Forgery (CSRF) was in the Top 10 for many years but has declined in importance due to improved defenses in web frameworks. CSRF attacks and defenses are covered in chapter 4.
2. g. Messages from John and all users' passwords will be returned from the query. This is known as an SQL injection UNION attack and shows that an attacker is not limited to retrieving data from the tables involved in the original query but can also query other tables in the database.
3. b. The attacker can get the program to allocate large byte arrays based on user input. For a Java `int` value, the maximum would be a 2GB array, which would probably allow the attacker to exhaust all available memory with a few requests. Although passing invalid values is an annoyance, recall from the start of section 2.5 that Java is a memory-safe language and so these will result in exceptions rather than insecure behavior.
4. d. `X-Content-Type-Options:` `nosniff` instructs browsers to respect the Content-Type header on the response.
5. b and e. You should never include unsanitized input values in error messages, as this may allow an attacker to inject XSS scripts. You should also never copy the Accept header from the request into the Content-Type header of a response, but instead construct it from scratch based on the actual content type that was produced.

Summary

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

---

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

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