spring by VMware Tanzu

< ALL GUIDES

Securing a Web Application

This guide walks you through the process of creating a simple web application with resources that are protected by Spring Security.

Get the Code

Go To Repo

Projects

Spring Security

What You Will Build

You will build a Spring MVC application that secures the page with a login form that is backed by a fixed list of users.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later
- Gradle 7.5+ or Maven 3.5+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
  - VSCode

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and c te each step or you can bypass basic setup steps that are already

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-securing-web.git`
- cd into `gs-securing-web/initial`
- Jump ahead to Create an Unsecured Web Application.

When you finish, you can check your results against the code in `gs-securing-web/complete`.

Starting with Spring Initializr

You can use this pre-initialized project and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web** and **Thymeleaf**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

or other editor.

Create an Unsecured Web Application

Before you can apply security to a web application, you need a web application to secure. This section walks you through creating a simple web application. Then you will secure it with Spring Security in the next section.

The web application includes two simple views: a home page and a “Hello, World” page. The home page is defined in the following Thymeleaf template (from `src/main/resources/templates/home.html`):

COPY

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="https://www.thymeleaf.org">
    <head>
        <title>Spring Security Example</title>
    </head>
    <body>
        <h1>Welcome!</h1>

        <p>Click <a th:href="@{/hello}">here</a> to see a
greeting.</p>
    </body>
</html>
```

This simple view includes a link to the `/hello` page, which is defined in the following Thymeleaf template (from `src/main/resources/templates/hello.html`):

COPY

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="https://www.thymeleaf.org">
    ead>
    <body>
        <h1>Hello world!</h1>
    </body>
</html>
```

The web application is based on Spring MVC. As a result, you need to configure Spring MVC and set up view controllers to expose these templates. The following listing (from `src/main/java/com/example/securingweb/MvcConfig.java`) shows a class that configures Spring MVC in the application:

COPY

```
package com.example.securingweb;

import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.ViewControlle
rRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigu
rer;

@Configuration
public class MvcConfig implements WebMvcConfigurer {

        public void addViewControllers(ViewControllerRegistry
registry) {

registry.addViewController("/home").setViewName("home");

registry.addViewController("/").setViewName("home");

registry.addViewController("/hello").setViewName("hello");

registry.addViewController("/login").setViewName("login");
        }
```

The `addViewControllers()` method (which overrides the method of the same name in `WebMvcConfigurer`) adds four view controllers. Two of the view controllers reference the view whose name is `home` (defined in `home.html`), and another references the view named `hello` (defined in `hello.html`). The fourth view controller references another view named `login`. You will create that view in the next section.

At this point, you could jump ahead to "Run the Application" and run the application without having to log in to anything.

Now that you have an unsecured web application, you can add security to it.

Set up Spring Security

Suppose that you want to prevent unauthorized users from viewing the greeting page at `/hello`. As it is now, if visitors click the link on the home page, they see the greeting with no barriers to stop them. You need to add a barrier that forces the visitor to sign in before they can see that page.

You do that by configuring Spring Security in the application. If Spring Security is on the classpath, Spring Boot automatically secures all HTTP endpoints with "basic" authentication. However, you can further customize the security settings. The first thing you need to do is add Spring Security to the classpath.

With Gradle, you need to add three lines (one for the application, one for Thymeleaf & Spring Security integration, and one for testing) in the `dependencies` closure in `build.gradle`, as the following listing shows:

COPY

```
implementation 'org.springframework.boot:spring-boot-starter-security'
//  Temporary explicit version to fix Thymeleaf bug
implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6:3.1.2.RELEASE'
```

The following listing shows the finished `build.gradle` file:

```
plugins {
	id 'java'
	id 'org.springframework.boot' version '3.3.0'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
	mavenCentral()
}

dependencies {
	implementation 'org.springframework.boot:spring-boot-starter-web'
	implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
	implementation 'org.springframework.boot:spring-boot-starter-security'
	//  Temporary explicit version to fix Thymeleaf bug
	implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6:3.1.2.RELEASE'
	testImplementation 'org.springframework.security:spring-security-test'
	testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
```

With Maven, you need to add two extra entries (one for the application and one for testing) to the `<dependencies>` element in `pom.xml`, as the following listing shows:

```
COPY
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
        <groupId>org.thymeleaf.extras</groupId>
        <artifactId>thymeleaf-extras-
springsecurity6</artifactId>
        <!-- Temporary explicit version to fix Thymeleaf bug --
>
        <version>3.1.1.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
</dependency>
```

The following listing shows the finished `pom.xml` file:

```
COPY
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-
parent</artifactId>
                <version>3.3.0</version>
                <relativePath/> <!-- lookup parent from
```

spring by VMware Tanzu

```
	<groupId>com.example</groupId>
	<artifactId>securing-web-complete</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>securing-web-complete</name>
	<description>Demo project for Spring Boot</description>

	<properties>
		<java.version>17</java.version>
	</properties>

	<dependencies>
		<dependency>

<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-
thymeleaf</artifactId>
		</dependency>
		<dependency>

<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-
web</artifactId>
		</dependency>
		<dependency>

<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-
security</artifactId>
		</dependency>
		<dependency>
			<groupId>org.thymeleaf.extras</groupId>
			<artifactId>thymeleaf-extras-
springsecurity6</artifactId>
			<!-- Temporary explicit version to fix
Thymeleaf bug -->
			<version>3.1.1.RELEASE</version>
		</dependency>
```

```
<groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-
test</artifactId>
                        <scope>test</scope>
                </dependency>

                <dependency>

<groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-
test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

</project>
```

The following security configuration (from `src/main/java/com/example/securingweb/WebSecurityConfig.java`) ensures that only authenticated users can see the secret greeting:

```
package com.example.securingweb;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.Htt
pSecurity;
import
org.springframework.security.config.annotation.web.configuratio
n.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsServic
```

```
nager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

        @Bean
        public SecurityFilterChain
securityFilterChain(HttpSecurity http) throws Exception {
                http
                        .authorizeHttpRequests((requests) ->
requests
                                .requestMatchers("/",
"/home").permitAll()
                                .anyRequest().authenticated()
                        )
                        .formLogin((form) -> form
                                .loginPage("/login")
                                .permitAll()
                        )
                        .logout((logout) ->
logout.permitAll());

                return http.build();
        }

        @Bean
        public UserDetailsService userDetailsService() {
                UserDetails user =
                         User.withDefaultPasswordEncoder()
                                .username("user")
                                .password("password")
                                .roles("USER")
                                .build();

                return new InMemoryUserDetailsManager(user);
```

The `WebSecurityConfig` class is annotated with `@EnableWebSecurity` to enable Spring Security's web security support and provide the Spring MVC integration. It also exposes two beans to set some specifics for the web security configuration:

The `SecurityFilterChain` bean defines which URL paths should be secured and which should not. Specifically, the `/` and `/home` paths are configured to not require any authentication. All other paths must be authenticated.

When a user successfully logs in, they are redirected to the previously requested page that required authentication. There is a custom `/login` page (which is specified by `loginPage()`), and everyone is allowed to view it.

The `UserDetailsService` bean sets up an in-memory user store with a single user. That user is given a user name of `user`, a password of `password`, and a role of `USER`.

Now you need to create the login page. There is already a view controller for the `login` view, so you need only to create the login view itself, as the following listing (from `src/main/resources/templates/login.html`) shows:

COPY

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="https://www.thymeleaf.org">
    <head>
        <title>Spring Security Example </title>
    </head>
    <body>
        <div th:if="${param.error}">
            Invalid username and password.
        </div>
        <div th:if="${param.logout}">
            You have been logged out.
        </div>
        <form th:action="@{/login}" method="post">
```

```
            <div><label> Password: <input type="password"
name="password"/> </label></div>
            <div><input type="submit" value="Sign In"/></div>
        </form>
    </body>
</html>
```

This Thymeleaf template presents a form that captures a username and password and posts them to `/login`. As configured, Spring Security provides a filter that intercepts that request and authenticates the user. If the user fails to authenticate, the page is redirected to `/login?error`, and your page displays the appropriate error message. Upon successfully signing out, your application is sent to `/login?logout`, and your page displays the appropriate success message.

Last, you need to provide the visitor a way to display the current user name and sign out. To do so, update the `hello.html` to say hello to the current user and contain a `Sign Out` form, as the following listing (from `src/main/resources/templates/hello.html`) shows:

COPY

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="https://www.thymeleaf.org"
      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-
springsecurity6">
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h1 th:inline="text">Hello <span th:remove="tag"
sec:authentication="name">thymeleaf</span>!</h1>
        <form th:action="@{/logout}" method="post">
            <input type="submit" value="Sign Out"/>
        </form>
    </body>
</html>
```

successfully logging out, it redirects the user to `/login?logout`.

Thymeleaf 3.1 no longer provides access to `HttpServletRequest` so `HttpServletRequest#getRemoteUser()` cannot be used to access the currently authenticated user.

Run the Application

The Spring Initializr creates an application class for you. In this case, you need not modify the class. The following listing (from `src/main/java/com/example/securingweb/SecuringWebApplication.java`) shows the application class:

COPY

```
package com.example.securingweb;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SecuringWebApplication {

        public static void main(String[] args) throws Throwable
{

SpringApplication.run(SecuringWebApplication.class, args);
        }

}
```

Build an executable JAR

Yo n run the application from the command line with Gradle or M . You can also build a single executable JAR file that contains all service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

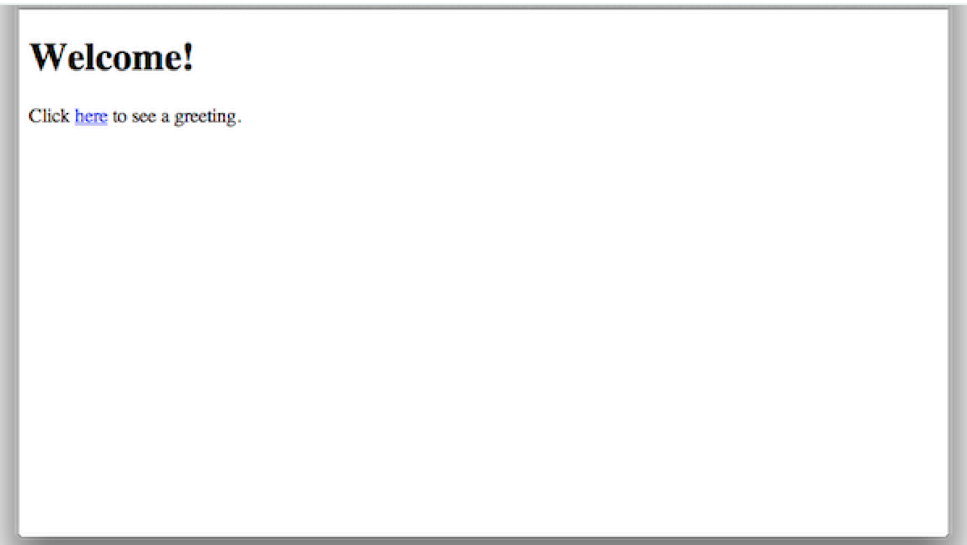
```
java -jar build/libs/gs-securing-web-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-securing-web-0.1.0.jar
```

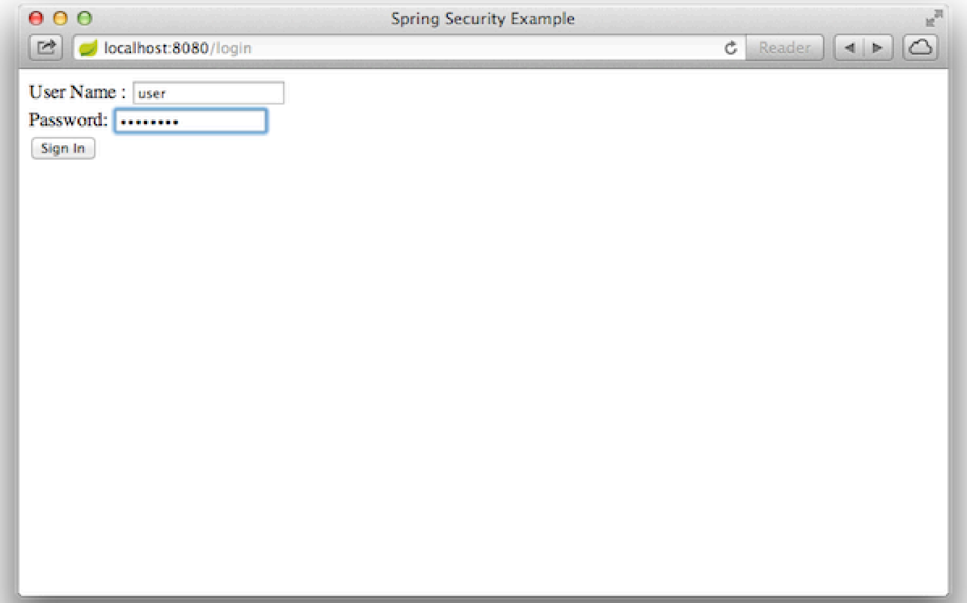
The steps described here create a runnable JAR. You can also build a classic WAR file.

Once the application starts up, point your browser to `http://localhost:8080`. You should see the home page, as the following image shows:


Welcome!

Click here to see a greeting.

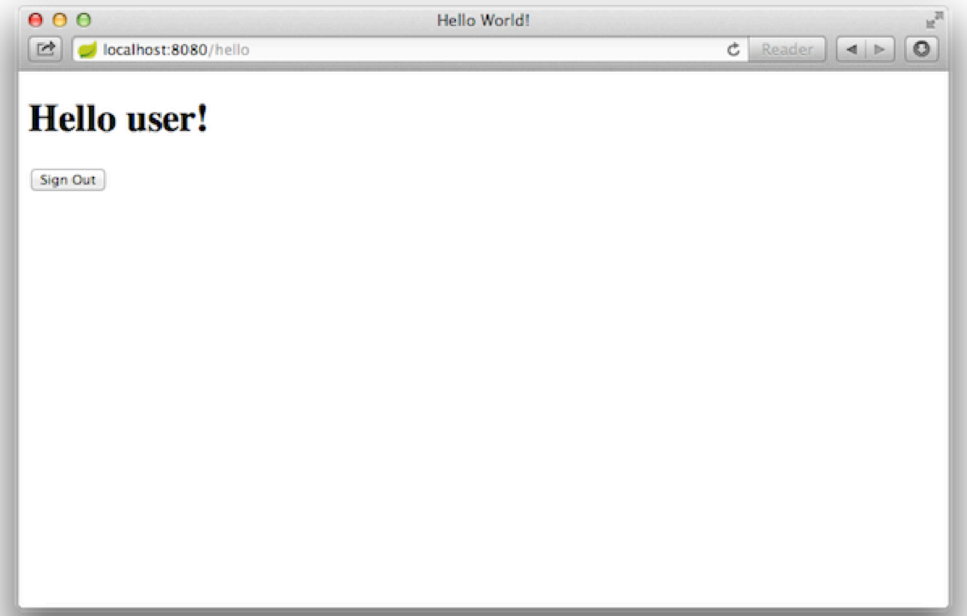
When you click on the link, it attempts to take you to the greeting page at `/hello`. However, because that page is secured and you have not yet logged in, it takes you to the login page, as the following image shows:



User Name : user
Password: ••••••••
Sign In

If you jumped down here with the unsecured version, you do not see the login page. You should back up and write the rest of the security-based code.

you submit the login form, you are authenticated and then taken to the greeting page, as the following image shows:



Hello user!

If you click on the **Sign Out** button, your authentication is revoked, and you are returned to the login page with a message indicating that you are logged out.

Summary

Congratulations! You have developed a simple web application that is secured with Spring Security.

See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Serving Web Content with Spring MVC
- Spring Security and Angular JS (Tutorial)

W[illegible] to write a new guide or contribute to an existing one? Check out o[illegible]tribution guidelines.

Attribution, NoDerivatives creative commons license for the writing.

Why Spring

Microservices

Reactive

Event Driven

Cloud

Web Applications

Serverless

Batch

Learn

Quickstart

Guides

Blog

Community

Events

Authors

Solutions

Tanzu Spring

Spring Consulting

Spring Academy For Teams

Spring Advisories

Projects

Training

Thank You

Get the Spring newsletter

Stay connected with the Spring newsletter

SUBSCRIBE