Form Login

Spring Security provides support for username and password being provided through an HTML form. This section provides details on how form based authentication works within Spring Security.

This section examines how form-based login works within Spring Security. First, we see how the user is redirected to the login form:

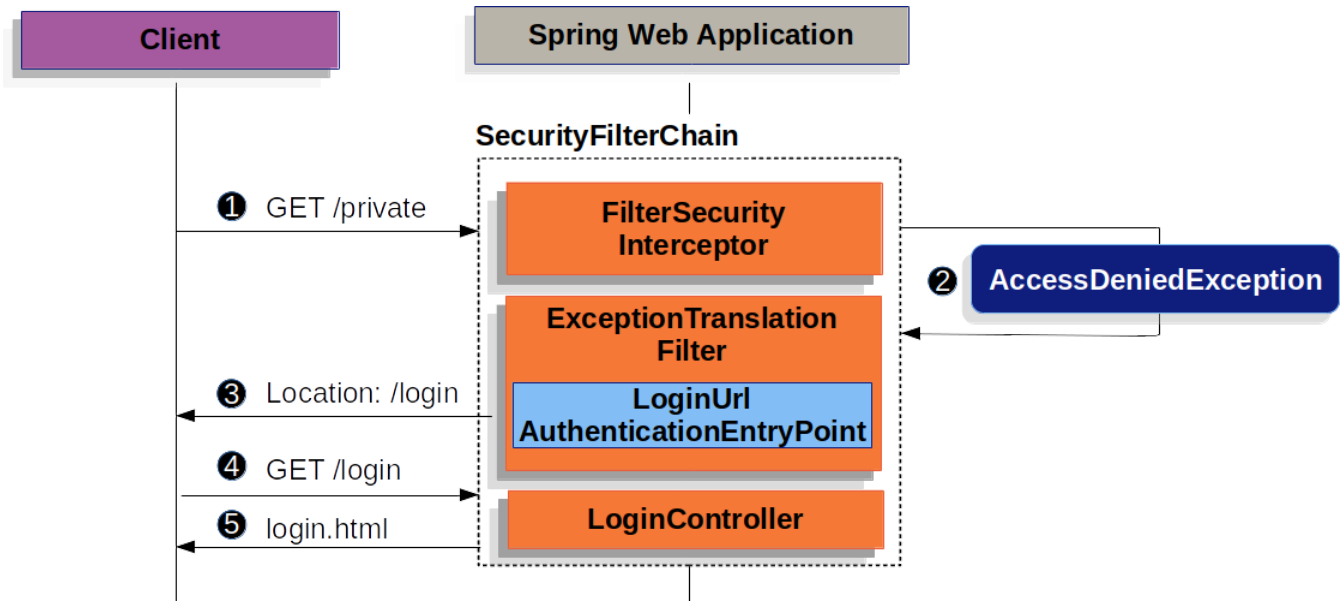

Figure 1. Redirecting to the Login Page

The preceding figure builds off our `SecurityFilterChain` diagram.

1 First, a user makes an unauthenticated request to the resource (`/private`) for which it is not authorized.

2 Spring Security's `AuthorizationFilter` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException` .

3 Since the user is not authenticated, `ExceptionTranslationFilter` initiates *Start Authentication* and sends a redirect to the login page with the configured `AuthenticationEntryPoint` . In most cases, the `AuthenticationEntryPoint` is an instance of `LoginUrlAuthenticationEntryPoint` .

4 The browser requests the login page to which it was redirected.

5 Something within the application, must render the login page.

When the username and password are submitted, the `UsernamePasswordAuthenticationFilter` authenticates the username and password. The `UsernamePasswordAuthenticationFilter` extends AbstractAuthenticationProcessingFilter, so the following diagram should look pretty similar:

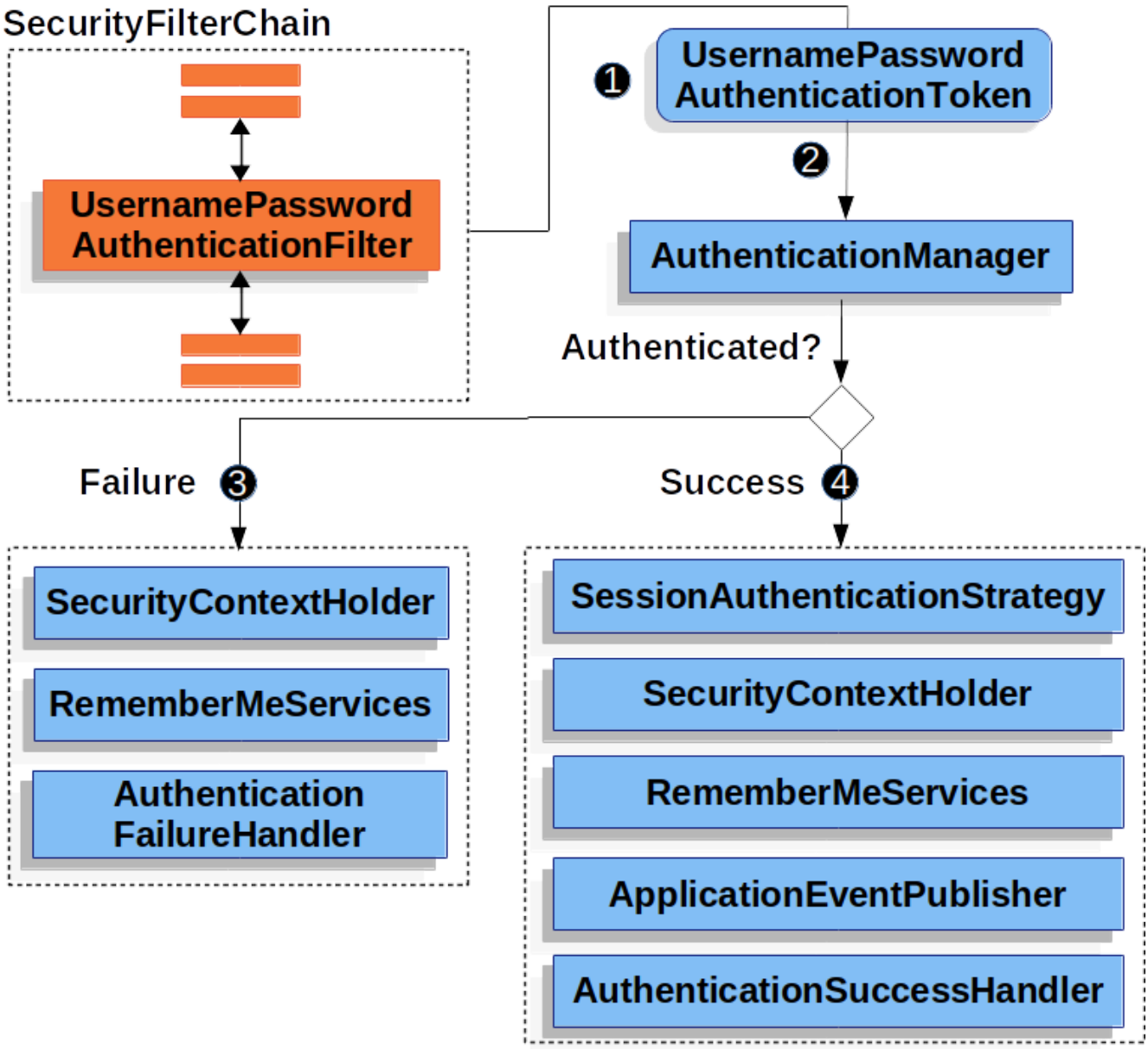

Figure 2. Authenticating Username and Password

The figure builds off our `SecurityFilterChain` diagram.

1 When the user submits their username and password, the `UsernamePasswordAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken`, which is a type of `Authentication`, by extracting the username and password from the `HttpServletRequest` instance.

2 Next, the `UsernamePasswordAuthenticationToken` is passed into the `AuthenticationManager` instance to be authenticated. The details of what `AuthenticationManager` looks like depend on how the user information is stored.

3 If authentication fails, then *Failure*.

1. The SecurityContextHolder is cleared out.
2. `RememberMeServices.loginFail` is invoked. If remember me is not configured, this is a no-op. See the `RememberMeServices` interface in the Javadoc.
3. `AuthenticationFailureHandler` is invoked. See the `AuthenticationFailureHandler` class in the Javadoc

4. If authentication is successful, then *Success*.

1. `SessionAuthenticationStrategy` is notified of a new login. See the `SessionAuthenticationStrategy` interface in the Javadoc.
2. The Authentication is set on the SecurityContextHolder. See the `SecurityContextPersistenceFilter` class in the Javadoc.
3. `RememberMeServices.loginSuccess` is invoked. If remember me is not configured, this is a no-op. See the `RememberMeServices` interface in the Javadoc.
4. `ApplicationEventPublisher` publishes an `InteractiveAuthenticationSuccessEvent`.
5. The `AuthenticationSuccessHandler` is invoked. Typically, this is a `SimpleUrlAuthenticationSuccessHandler`, which redirects to a request saved by `ExceptionTranslationFilter` when we redirect to the login page.

By default, Spring Security form login is enabled. However, as soon as any servlet-based configuration is provided, form based login must be explicitly provided. The following example shows a minimal, explicit Java configuration:

Form Login

Java | **XML** | **Kotlin**

```java
public SecurityFilterChain filterChain(HttpSecurity http) {
    http
        .formLogin(withDefaults());
    // ...
}
```

In the preceding configuration, Spring Security renders a default login page. Most production applications require a custom login form.

The following configuration demonstrates how to provide a custom login form.

Custom Login Form Configuration

Java | **XML** | **Kotlin**

```java
public SecurityFilterChain filterChain(HttpSecurity http) {
    http
        .formLogin(form -> form
            .loginPage("/login")
```

```
            .permitAll()
        );
    // ...
}
```

When the login page is specified in the Spring Security configuration, you are responsible for rendering the page. The following Thymeleaf template produces an HTML login form that complies with a login page of `/login`.:

Login Form - src/main/resources/templates/login.html

```xml
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
    <head>
        <title>Please Log In</title>
    </head>
    <body>
        <h1>Please Log In</h1>
        <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>
            <input type="text" name="username" placeholder="Username"/>
            </div>
            <div>
            <input type="password" name="password" placeholder="Password"/>
            </div>
            <input type="submit" value="Log in" />
        </form>
    </body>
</html>
```

There are a few key points about the default HTML form:

- The form should perform a `post` to `/login`.
- The form needs to include a CSRF Token, which is automatically included by Thymeleaf.
- The form should specify the username in a parameter named `username`.
- The form should specify the password in a parameter named `password`.
- If the HTTP parameter named `error` is found, it indicates the user failed to provide a valid username or password.
- If the HTTP parameter named `logout` is found, it indicates the user has logged out successfully.

Many users do not need much more than to customize the login page. However, if needed, you can customize everything shown earlier with additional configuration.

If you use Spring MVC, you need a controller that maps `GET /login` to the login template we created. The following example shows a minimal `LoginController`:

LoginController

Java | Kotlin

```java
@Controller
class LoginController {
    @GetMapping("/login")
    String login() {
        return "login";
    }
}
```