# Programming 5

Security

KdG Karel de Grote Hogeschool

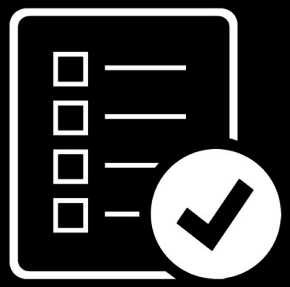
# Schedule for term 3

| | |
|---|---|
| Week 1 | Project setup and refactoring |
| Week 2 | Web API<br>● REST and AJAX<br>● Implementation using Spring framework<br>● Frontend: fetch |
| Week 3 | |
| Week 4 | Security<br>● Form login, cookies, CSRF, …<br>● Implementation using Spring framework<br>● Users & Roles |
| Week 5 | |
| Week 6 | Backend Testing<br>● JUnit<br>● Integration testing with Spring |

# Schedule for term 4

| | |
|---|---|
| Week 7 | <u>Backend Testing</u><br>● Mocking (unit testing)<br>● Testing Spring Security |
| Week 8 | |
| Week 9 | <u>Frontend</u><br>● Frontend: npm and webpack<br>● Frontend build step<br>● Bundling, transpilation, … |
| Week 10 | |
| Week 11 | |
| Week 12 | ● Asynchronous processing<br>● File uploads<br>● Cleanup … |

- Enabling Spring Security
- Controlling Access
- Handling Login
- Stateful Authentication
  - Cookies and CSRF
- Users and Roles

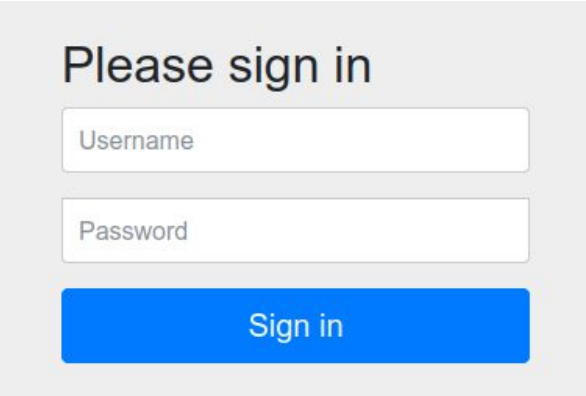


# Gradle setup

- **`build.gradle`:**

```
...
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-security'
  implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6'
  ...
}
...
```

- Enable Spring Security with *default* settings and Thymeleaf support




Everything's locked tight!
Standard login page.

# Spring Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http.authorizeHttpRequests(
                auths -> auths
                        .requestMatchers(regexMatcher("^/issue\\?.+"))
                            .permitAll()
                        .requestMatchers(
                                antMatcher(HttpMethod.GET, "/js/**"),
                                antMatcher(HttpMethod.GET, "/css/**"))
                            .permitAll()
                        .requestMatchers(antMatcher(HttpMethod.GET, "/"))
                               .permitAll()
                        .anyRequest()
                            .authenticated()
            )
            .formLogin(formLogin -> formLogin.permitAll());
        return http.build();
    }
}
```

# Spring Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http.authorizeHttpRequests(
                auths -> auths
                        .requestMatchers(
                            .permitAll()
                        .requestMatchers(
                                antMatcher(HttpMethod.GET, "/js/**"),
                                antMatcher(HttpMethod.GET, "/css/**"))
                            .permitAll()
                        .requestMatchers(antMatcher(HttpMethod.GET, "/"))
                              .permitAll()
                        .anyRequest()
                            .authenticated()
            )
            .formLogin(formLogin -> formLogin.permitAll());
        return http.build();
    }
}
```

Enable web security in a configuration class using **`@EnableWebSecurity`**. Provide a bean of type **`SecurityFilterChain`**.

# Spring Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http.authorizeHttpRequests(
                auths -> auths
                        .requestMatchers(regexMatcher("^/issue\\?.+"))
                            .permitAll()
                        .requestMatchers(
                                antMatcher(HttpMethod.GET, "/js/**"),
                                antMatcher(HttpMethod.GET, "/css/**"))
                            .permitAll()
                        .requestMatchers(antMatcher(HttpMethod.GET, "/"))
                                .permitAll()
                                                                    mitAll());
        return http.build();
    }
}
```

Disable security for static resources and/or large subsections of the web application.
(Not just **js** and **css**, this is just an example!)

# Spring Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http.authorizeHttpRequests(
                auths -> auths
                        .requestMatchers(r
                            .permitAll()
                        .requestMatchers(
                                antMatcher
                                antMatcher
                            .permitAll()
                        .requestMatchers(antMatcher(HttpMethod.GET, "/"))
                               .permitAll()
                        .anyRequest()
                            .authenticated()
            )
            .formLogin(formLogin -> formLogin.permitAll());
        return http.build();
    }
}
```

- Permit access to '**/**' and possibly other pages ( '**/register**' perhaps?).
- Require authentication for any other request.

# Spring Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http.authorizeHttpRequests(
                auths -> auths
                        .requestMatchers(regexMatcher("^/issue\\?.+"))
                            .permitAll()
                        .requestMatchers(
                                antMatcher(HttpMethod.GET, "/js/**"),
                                antMatcher(HttpMethod.GET, "/css/**"))
                            .permitAll()
                        .requestMatchers(antMatcher(HttpMethod.GET, "/"))
                            .permitAll()
                        .anyRequest()
                            .authenticat
            )
            .formLogin(formLogin -> formLogin.permitAll());
        return http.build();
    }
}
```

Enable the *default* form login page at '**/login**'. (the one we saw earlier).

# Spring Configuration

- Introduction to Java Config for Spring Security:
  - https://www.baeldung.com/java-config-spring-security

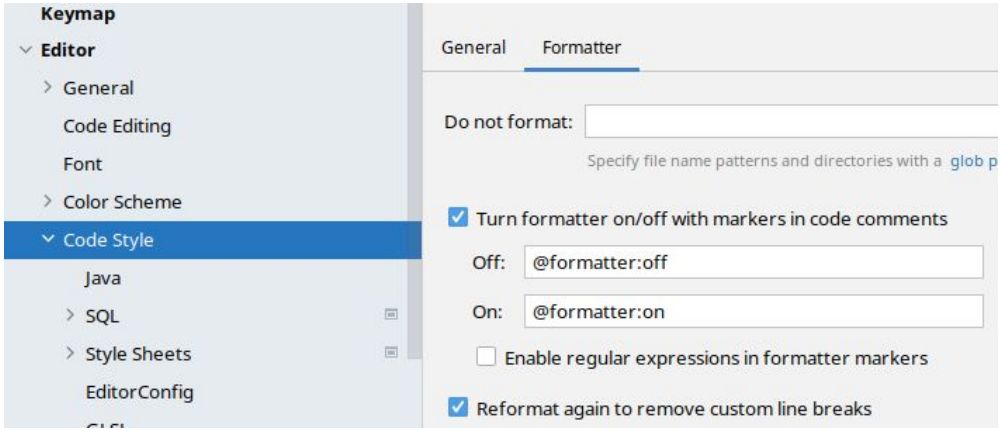
Unfortunately, in this document, they are still using Spring Security version 5.
The API has changed quite a bit…

Baeldung

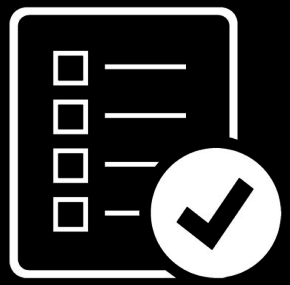
# Spring Configuration

- To keep things readable, you may want to turn off the formatter temporarily:

```
// @formatter:off
http.authorizeHttpRequests(
    auths -> auths
        .requestMatchers( ... )
            .permitAll()
        .requestMatchers(
        ...
    )
    .formLogin(formLogin -> formLogin.permitAll());
// @formatter:on
```



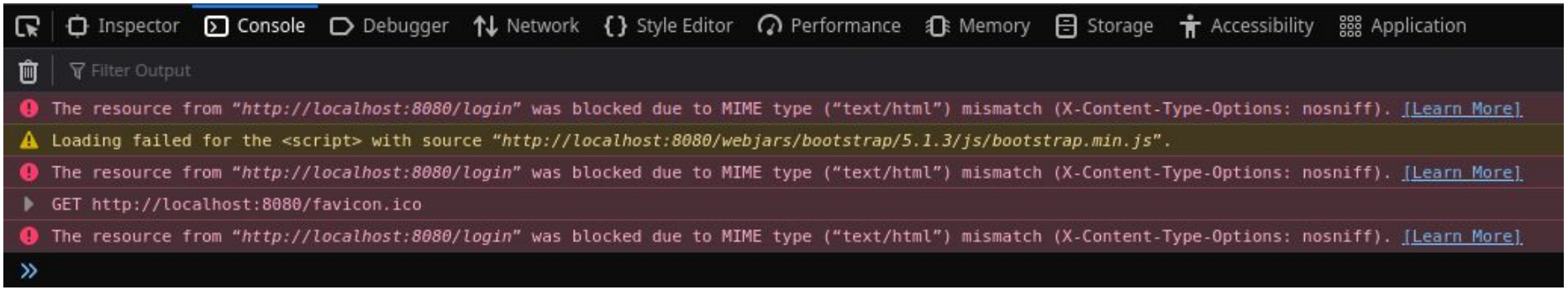
- Enabling Spring Security
- Controlling Access
- Handling Login
- Stateful Authentication
  - Cookies and CSRF
- Users and Roles



# Controlling access

- CSS and JS files need to be permitted explicitly, or else your app will break:

```
.requestMatchers(
        antMatcher(HttpMethod.GET, "/js/**"),
        antMatcher(HttpMethod.GET, "/css/**"),
        antMatcher(HttpMethod.GET, "/webjars/**"),
        regexMatcher(HttpMethod.GET, ".+\\.ico"))
    .permitAll()
```




Forgetting about JS and CSS will trigger these confusing error messages, why? 🤔

# Controlling access

- Specify paths/endpoints using **matchers**
- Some options:

  **`regexMatcher`**

  **`antMatcher`**

- Special case:

  **`anyRequest`**

- Matches are considered **in order**

Matches any other request not yet covered. To be placed **last**!

# Controlling access

- **regexMatcher**

```
.requestMatchers(
        regexMatcher("^/(issue\\?.+|issues)"))
```

- **antMatcher**

```
.requestMatchers(
        antMatcher(HttpMethod.GET, "/api/**"))
```

'Ant' is an ancient build tool from the 1990's.

Matcher info.

# Controlling access

- No need to be authenticated:

```
.permitAll()
```

- Must be authenticated:

```
.authenticated()
```

- Not accessible by anyone:

```
.denyAll()
```

- Must follow a matcher:

```
.regexMatchers("^/(issue\\?.+|issues)")
   .permitAll()
.antMatchers(HttpMethod.GET, "/api/**")
   .permitAll()
.antMatchers("/", "/register")
   .permitAll()
```

Chained together.

# Controlling access

- Intro to Spring Security Expressions:
  - https://www.baeldung.com/spring-security-expressions

Unfortunately, in this document, they are still using Spring Security version 5.
The API has changed quite a bit…

# Login and Logout options

- We'll use an HTML form to log in
  - Enable with the *default* login form:

```
.formLogin(formLogin ->
    formLogin.permitAll())
```

  - Specify a custom login page:

```
.formLogin(formLogin ->
    formLogin
        .loginPage("/login")
        .permitAll()
)
```

  - In **both** cases, the POST action for handling the form submit is provided by Spring

# Login and Logout options

- There’s no ‘log out’ page, it’s usually just a button:

```
.logout(logout -> logout.permitAll())
```

  - The POST action for handling the form submit is provided by Spring

# Login and Logout options

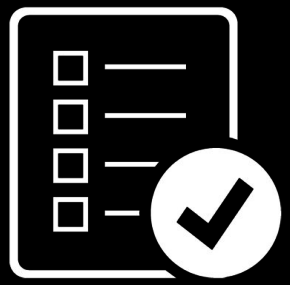
- Problem: Try accessing a secured **REST** endpoint with invalid credentials: redirect (302) followed by an OK (200) ⇒ redirect to login page
- Solution:

```
.exceptionHandling(exceptionHandling ->
    exceptionHandling.authenticationEntryPoint(
        (request, response, exception) -> {
            if (request.getRequestURI().startsWith("/api")) {
                response.setStatus(HttpStatus.FORBIDDEN.value());
            } else {
                response.sendRedirect(
                        request.getContextPath() + "/login");
            }
        })
)
```

# Example

```
http
    .authorizeHttpRequests(
        auths ->
            auths
                . /* more config here */
    )
    .formLogin(formLogin ->
        formLogin
            .loginPage("/login")
            .permitAll()
    )
    .exceptionHandling(exceptionHandling ->
        exceptionHandling.authenticationEntryPoint(
            (request, response, exception) -> {
                if (request.getRequestURI().startsWith("/api")) {
                    response.setStatus(HttpStatus.FORBIDDEN.value());
                } else {
                    response.sendRedirect(request.getContextPath() + "/login");
                }
            })
    );
```

- Enabling Spring Security
- Controlling Access
- **Handling Login**
- **Stateful Authentication**
  - **Cookies and CSRF**
- **Users and Roles**



# Handling login

- Enable a custom login form:

```
.formLogin(formLogin ->
   formLogin
       .loginPage("/login")
       .permitAll())
```

- Let's make the form available:

```
@GetMapping("/login")
public String showLogin() {
   return "login";
}
```

- Create the view:

```
<!DOCTYPE html>
<html lang="en"
     xmlns:th="http://www.thymeleaf.org"
     ...
```

# Handling login

- As mentioned, we **don't** need to write the `@PostMapping("/login")` method!

- … but then what does the form need to submit?
  - **`username`**
  - **`password`**

Plain HTML code is fine:
**`name="username"`** etc.
… but we can also use Thymeleaf and a viewmodel.

- It **has** to be *username* and *password*, but a username can be anything (even an email address!)

Once deployed, **HTTPS** will take care of encryption!

# Handling login

- Spring looks for a service that implements **`UserDetailsService`**.
  We provide our custom implementation:

```
@Service
public class CustomUserDetailsService
        implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String user)
            throws UsernameNotFoundException {
        ...
    }
}
```

# Handling login

- **`UserDetailsService`**
  Make a component that implements this interface; ensure it's available through dependency injection

- Implement **`loadUserByUsername`**:
  - Retrieve the user
  - Return the user as a **`UserDetails`** object

This is Spring's interface for representing the users of an application.

# Handling login

- Extend Spring's **`User`** class (which implements **`UserDetails`**) to add application-specific properties:

```
public class CustomUserDetails extends User {
    private final long userId;

    . . .
}
```

You can then return this custom user from the **`loadUserByUsername`** method!

- **Important!** This is a "security framework" user, **<u>not</u>** a domain User!

# Handling login

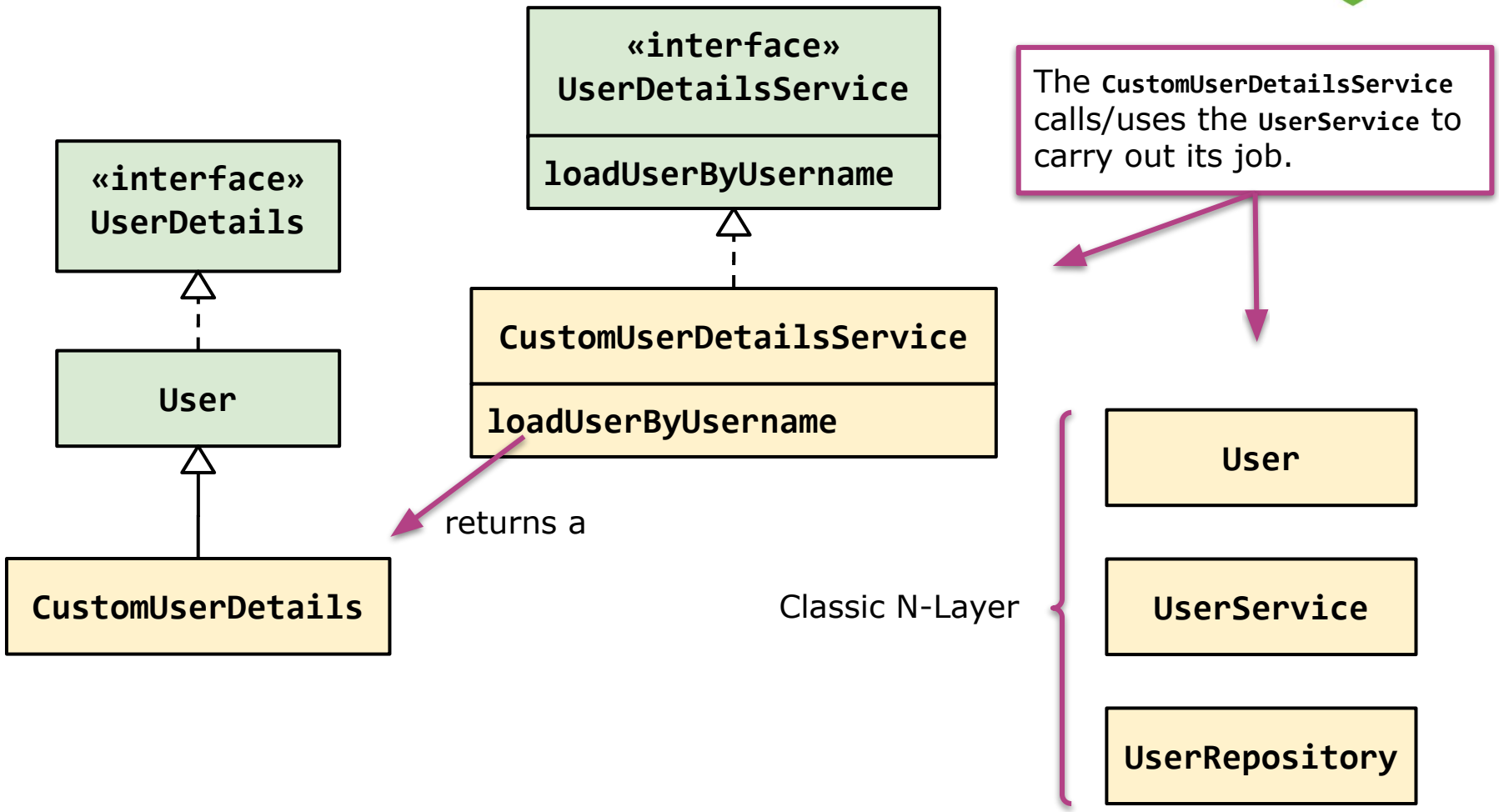
- We'll also need a user in the domain:

```
@Entity
@Table(name = "application_user")
public class User {
    @Id
    @GeneratedValue
    ...
```

The default table name 'user' is an SQL reserved keyword.
Table name 'user' will cause issues possibly *without* any clear error messages!

- A **`UserRepository`** and a **`UserService`** (in addition to the **`CustomUserDetailsService`**!) make a lot of sense now …

# Handling login



# Sign in automatically

```
@PostMapping("register")
public String registerNewUser(
        @Valid NewUserViewModel userViewModel,
        HttpServletRequest request) throws ServletException {
    userService.createNewUser(
            userViewModel.getUsername(),
            userViewModel.getPassword1());
    request.login(
            userViewModel.getUsername(),
            userViewModel.getPassword1());
    return "redirect:/";
}
```

# Password hashing

- Use of a password encoder is **mandatory** when using Spring Security
  - We'll use the BCrypt hash function
  - Seeding: https://www.browserling.com/tools/bcrypt

```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- Registration with Spring Security – Password Encoding
  - Baeldung article

# Thymeleaf

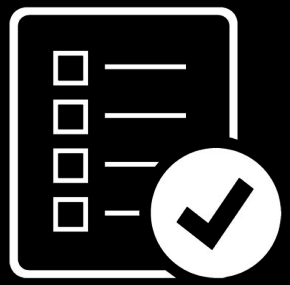
- Documentation
- GitHub repository
- The Gradle dependency has been added

```
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
      ...

    <div sec:authorize="!isAuthenticated()">
        ...
    </div>
    <div sec:authorize="isAuthenticated()">
        <span sec:authentication="name"></span>
    </div>
...
```

In the `sec:authorize` attribute, you need to use Spring Security Expressions

- Enabling Spring Security
- Controlling Access
- Handling Login
- **Stateful Authentication**
  - **Cookies and CSRF**
- **Users and Roles**



# Stateful Authentication

- Server-side state / session that keeps track of who we are
- Typically Cookie-based

Key/value pair (text)

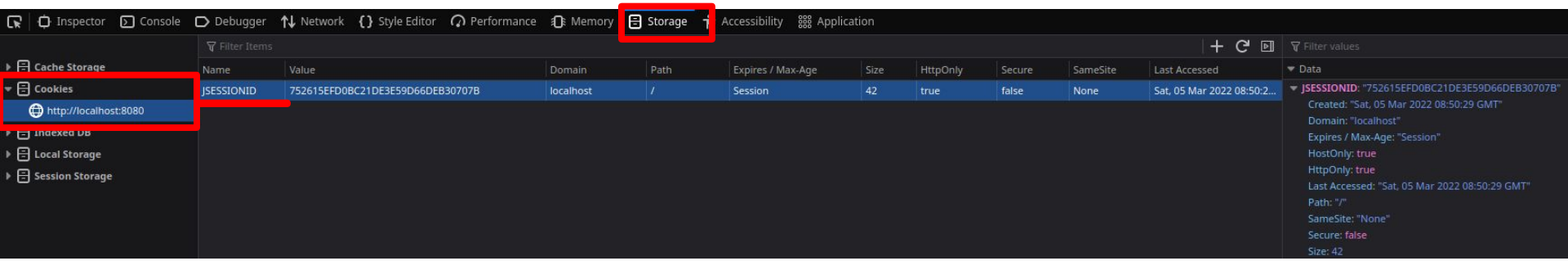
Automatically transmitted with each subsequent request

HTTP Headers **`Set-Cookie`** and **`Cookie`**

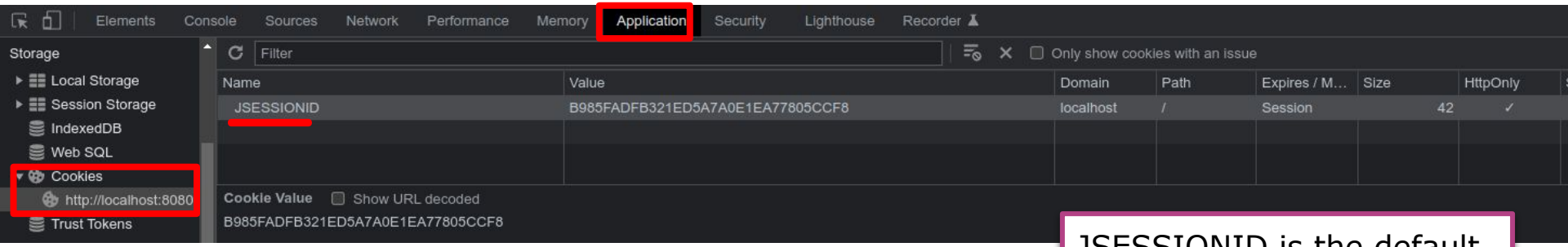
Stored in the browser for this site (domain only)

# Stateful Authentication

- Firefox:




- Chrome:




JSESSIONID is the default cookie name of a stateful session in Java web applications.

# Stateful Authentication

- While navigating our application, we remain logged in:
  - We keep sending the cookie on *every* http request (click, fetch with AJAX, form submit, …)
  - The server remembers us (cookie is tied to a session, managed by Spring)
- Very convenient, but introduces a load of potential security issues

**https://bank.com/transmit?user=Lars&amount=1000000000**

Trick someone into clicking this link (fake email). If that person has a cookie (=authenticated session), it is automatically transmitted!

# Transmitting cookies

- Signing in and obtaining a cookie

We want to observe all HTTP traffic!

We want to manage cookies **manually**!

HTTP Request

```
# @no-redirect
# @no-cookie-jar
POST http://localhost:8080/login
Accept: text/html
Content-Type: application/x-www-form-urlencoded

username=lars.willemsens@kdg.be&password=lars
```

# Transmitting cookies

- Signing in and obtaining a cookie

<u>HTTP Response</u>

```
HTTP/1.1 302
Set-Cookie: JSESSIONID=9EF...291; Path=/; HttpOnly
Location: http://localhost:8080/
Content-Length: 0
# Other headers have been omitted!

<Response body is empty>
```

This is the actual cookie.

"302 Found" is a redirection response.
Check the Location URL for

- **success** (http://localhost:8080/
- … or **failure** (http://localhost:8080/login?error)

# Transmitting cookies

- Using the cookie in subsequent requests

HTTP Request

```
# @no-redirect
# @no-cookie-jar
GET http://localhost:8080/book/all
Accept: text/html
Cookie: JSESSIONID=9EF...291
```

In the HTML body of the response I'll be able to see my name and a "Log out" button.

# Transmitting cookies

- Manually storing the cookie in a variable in your IDE

HTTP Request

```
# @no-redirect
# @no-cookie-jar
POST http://localhost:8080/login
Accept: text/html
Content-Type: application/x-www-form-urlencoded

username=lars.willemsens@kdg.be&password=lars

// Save the cookie to reuse it later
> {%
   client.global.set(
       "spring_cookie",
       response.headers.valuesOf("Set-Cookie")
           .find(cookie => cookie.includes("JSESSIONID"))
           .split(';')[0]
           .split('=')[1]
   );
%}
```

# Transmitting cookies

- Manually storing the cookie in a variable in your IDE

<u>Subsequent HTTP Request</u>

```
# @no-redirect
# @no-cookie-jar
GET http://localhost:8080/book/all
Accept: text/html
Cookie: JSESSIONID={{spring_cookie}}
```

# CSRF - In a nutshell

**CSRF (or XSRF): Cross-Site Request Forgery**

1. From an attacker's website
2. Tricking users to:
   - Click on a link
   - (automatically) submit a form
   - Send an AJAX request with fetch API
3. That performs an action on **another** website
4. While the user is authenticated on that other website

# CSRF - In a nutshell

https://bannk.com

Fake Bank website

FORM
POST
action="https://bank.com"

Field 1
Field 2
send funds to lars

**Cookie**
Sent by browser
(NOT by attacker)

No CSRF token?
No succes!

https://bank.com

Real Bank website

<POST arrives here>

Solution: Add CSRF token to this page

# CSRF - How to fix

- A web application will provide additional information *(a "CSRF token")* that is to be transmitted back to the server as something "*other than a cookie*" …
- Our approach: The CSRF token will be embedded in the web page and transmitted back to the server as an HTTP header

spring by Pivotal

Spring CSRF Docs

# CSRF - Attack surface

- GET requests
- Forms
- AJAX requests

# CSRF - GET requests

- We shouldn’t do anything to protect GET requests
- GET requests are already supposed to be safe

`https://bank.com/transmit?user=Lars&amount=1000000000`

Don’t implement an URL like this!
GET requests aren’t meant for actions like these (and shouldn’t have side effects).

# CSRF - Forms

- Forms in Spring MVC are handled automatically
- Verify using your browser's development tools:

```
▸<ul class="navbar-nav me-auto">…</ul> flex
▾<ul class="navbar-nav">  flex
  ▾<li class="nav-item">
    ▾<form id="searchForm" action="/book/search" method="post">
      <input type="hidden" name="_csrf" value="ae8f6b52-f5a0-4205-8b46-5f2e70b2a527">
      <input id="searchTerm" class="form-control me-2" name="searchTitle" type="search" p
      autocomplete="off">  event
```

CSRF Protection with Spring

Baeldung

# CSRF - AJAX requests

- By default, CSRF is enabled by Spring Security
- But the token itself is **not** yet made available in Javascript for AJAX requests
- Step 1: Make the CSRF token available to Javascript

  In thymeleaf:

```
<meta name="_csrf"
      th:content="${_csrf.token}"/>
<meta name="_csrf_header"
      th:content="${_csrf.headerName}"/>
```

Spring documentation

# CSRF - AJAX requests

- Step 2: Get the token in Javascript

  In your JavaScript source file:

```
const header = document
    .querySelector( ... );

const token = document
    .querySelector( ... );
```

# CSRF - AJAX requests

- Step 3: Add the token to a specific HTTP header

  Still in your JavaScript source file:

```
const headers = {};
headers[header] = token;

fetch('/api/publishers', {
   method: /* ... */,
   headers,
   body: /* ... */
})
```

# CSRF - AJAX requests

- The actual resulting HTTP request …
  - The CSRF token is transmitted in a specific header:

```
POST http://localhost:8080/api/publishers HTTP/1.1
Content-Type: application/json
Cookie: JSESSIONID=C73...B3C
X-XSRF-TOKEN: 152...7eb

{
   "name": "Lars Publishing inc.",
   "yearFounded": 1980
}
```

# CSRF

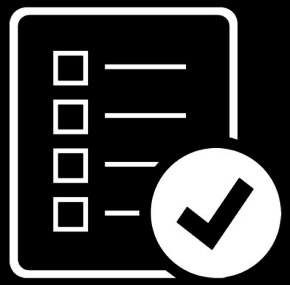
- While troubleshooting, you may want to disable CSRF **temporarily** (!) …

```
http.csrf(
    csrf -> csrf.disable()
)
```

- A Guide to CSRF Protection in Spring Security
  - https://www.baeldung.com/spring-security-csrf

Baeldung

- Enabling Spring Security
- Controlling Access
- Handling Login
- Stateful Authentication
  - Cookies and CSRF
- **Users and Roles**



# Accessing user information

- Any (REST-)Controller method can be annotated to capture user information:

```
@PostMapping
public ResponseEntity<BookDto> createBook(
      @RequestBody NewBookDto newBookDto,
      @AuthenticationPrincipal CustomUserDetails userDetails) {
  // ...
}
```

- Is **`null`** if the user isn't logged in
- Use a *custom* userdetails class to gain access to your application's custom user fields

# Roles

- As with users, roles can be looked at from two different perspectives:
  - **Spring**:
    - The role is typically represented by its name (= a String)
  - **Domain**:
    - Managed by your application code
    - Can be an @Entity, a boolean, a String, an enum, …

# Roles

- Roles are managed by our application code and then revealed to Spring:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String email) {
        ...
        final List<GrantedAuthority> authorities = new ArrayList<>();
        authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
        return new CustomUserDetails(/* ... */, authorities);
    }
}
```

This is the most basic case: hard-coded 'user' role.

- Role names **must start** with the **ROLE_** prefix!

# Roles

- There are many possible implementations
  - Call a RoleService?
    - Each user has a role? ManyToOne?
  - Add an **`isAdmin`** (boolean) attribute to User?
  - Add a **`role`** (enum) attribute to User?
  - ...

```
@Override
public UserDetails loadUserByUsername(String email) {
    ...
    final List<GrantedAuthority> authorities = new ArrayList<>();
    ...
    return new CustomUserDetails(/* ... */, authorities);
}
```

At the end of the day, we have to fill the list with some String-based **`SimpleGrantedAuthority`** objects.

# Roles

- In Thymeleaf we can easily check the role:

```
<div sec:authorize="hasRole('ROLE_ADMIN')">
   <!-- The div element will not be
        rendered for regular users. -->
</div>
```

In the `sec:authorize` attribute, you need to use Spring Security Expressions

- We could be hiding some information, a form, or an element with some JavaScript attached to it (a fetch call, perhaps)
- It's **essential** to also check for the role in the controller method that's called from the form or the fetch call!

Why?

# Roles

- Spring's **Method Security** provides a mechanism to easily check for roles
- It must be enabled using a configuration class:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
   // ...
}
```

# Roles

- Methods can now be annotated with, for example, **@PreAuthorize**:

```
@PostMapping
@PreAuthorize("hasRole('ROLE_ADMIN')")
public ResponseEntity<PublisherDto> createNewPublisher(
        @RequestBody @Valid NewPublisherDto publisherDto) {
    ...
```

In the `@PreAuthorize` annotation, you need to use Spring Security Expressions

- Keep these annotations limited to your **controllers**! Let them act as gatekeepers …

# Roles

- We can make our code more expressive using custom annotations:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('ROLE_ADMIN')")
public @interface AdminOnly {
}


@PostMapping
@AdminOnly
public ResponseEntity<PublisherDto> createNewPublisher(
        @RequestBody @Valid NewPublisherDto publisherDto) {
    ...
```

# Roles

- Spring Security: Check If a User Has a Role in Java
  - https://www.baeldung.com/spring-security-check-user-role

- Introduction to Spring Method Security
  - https://www.baeldung.com/spring-security-method-security