**Spring Security**

**Filters and Interceptors**

1. **Filters**

* A filter is an object used to **intercept** the HTTP **requests** and **responses** of your application
* We can perform two operations at two instances:
  + Before sending the **request** to the controller
  + Before sending a **response** to the client

**Filter Example**

**@Component**

public class GreetingFilter implements Filter {

**@Override**

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,

FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletResponse;

request.getSession().setAttribute('name', 'Pesho');

filterChain.doFilter(request, response);

}

}

**@Controller**

public class HomeController {

**@GetMapping('/')**

public ModelAndView index(ModelAndView modelAndView, HttpSession session) {

modelAndView.setViewName('index');

modelAndView.addObject('name', session.getAttribute('name'));

return modelAndView;

}

}

1. **Interceptor**

* A **Filter** is used in the **web layer only** as it is defined in web.xml. We can not use it out of web context
* While Spring **Interceptors** are defined in the Spring context
* The **interceptor** include **three** main **methods**:
  + **preHandle**: executed before the execution of the target resource
  + **afterCompletion:** executed after the execution of the target resource (after rendering the view)
  + **postHandle:** Intercept the execution of a handler

**Interceptor Example**

public class LogingInteceptor implements HandlerInterceptor {

**@Override**

public boolean preHandle(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain, Object handler) throws IOException, ServletException {

//Log some information ...

retrun true;

}

}

**Register Interceptor in Configuration**

To use interceptors we need to register them

**@Configuration**

public class WebConfiguration implements WebMvcConfigurer {

private final MyInterceptor myInterceptor;

public WebConfiguration(MyInterceptor myInterceptor) {

this.myInterceptor = myInterceptor;

}

**@Override**

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(myInterceptor);

}

}

1. **What is Spring Security?**

* A **powerful** and **highly customizable** authentication and access-control framework
* It is the de-facto **standard** for securing **Spring-based** applications
* Focuses on providing both **authentication** and **authorization** to Java applications
  1. **Authentication**

Who is logged in ?

* 1. **Authorization**

What you are allowed to do ?

**Security Context and Authentication**

****

* At the heart of Spring Security's authentication model is the **SecurityContextHolder**
* It contains the **SecurityContext**

**Spring Security Maven/Gradle**

* Adding **Spring Security**.

**Maven**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**Gradle**

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-security'

}

**Spring Security Configuration**

* We need to implement **UserDetails** interface

public interface UserDetails {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}

* Implementing the **GrantedAuthority** interface.

public class Role implements GrantedAuthority {

private String authority;

}

**SimpleGrantedAuthority**

* If we want, we can use **SimpleGrantedAuthority** instead of creating Role class
* It is a basic concrete **implementation** of a **GrantedAuthority**
* Stores a **String representation** of an **authority** granted to the Authentication object
* Implementing the **UserDetailsService** interface.

public class UserDetailsServiceImpl implements **UserDetailsService** {

public UserDetailsServiceImpl() {

…

}

**@Override**

public UserDetails loadUserByUserName(String userName) {

// get the user and map to **UserDetails**

}  
}

* Expose as beans

**@Configuration**

public class SecurityConfiguration {

**@Bean**

public UserDetailsService userDetailsService(UserRepository

userRepository) {

return new UserDetailsServiceImpl(userRepository);

}

**@Bean**

public PasswordEncoder passwordEncoder() {

return new Pbkdf2PasswordEncoder();

}

**@Bean**

public SecurityFilterChain filterChain(HttpSecurity httpSecurity){

http

.authorizeRequests()

.antMatchers('/', '/register').permitAll()

.anyRequest().authenticated();

…

return http.build();

}

}

**Principal**

* This is the currently logged user

**Pre / Post Authorize**

* Grant Access to specific methods

1. **Cross-Site Request Forgery (CSRF)**

**Spring CSFR Protection**

.csrf()

.csrfTokenRepository(csrfTokenRepository())

private CsrfTokenRepository csrfTokenRepository() {

HttpSessionCsrfTokenRepository repository = new HttpSessionCsrfTokenRepository();

repository.setSessionAttributeName("\_csrf");

return repository;

}

<input type='hidden' th:name='${\_csrf.parameterName}' th:value='${\_csrf.token}' />

1. **Thymeleaf Security**

**Dependency**

<dependency>

<groupId>org.thymeleaf.extras</groupId>

<artifactId>thymeleaf-extras-springsecurity5</artifactId>

</dependency>

**Example**

<!DOCTYPE html>

<html lang='en'

xmlns:th='http://www.thymeleaf.org'

xmlns:sec='http://www.thymeleaf.org/extras/spring-security'>

<body>

<div sec:authentication='name'>

The value of the 'name' property of the authentication object should appear here.

</div>

</body>

</html>

<!DOCTYPE html>

<html lang='en'

xmlns:th='http://www.thymeleaf.org'

xmlns:sec='http://www.thymeleaf.org/extras/spring-security'>

<body>

<div sec:authorize='hasRole('ADMIN')'>

This content is only shown to administrators.

</div>

</body>

</html>

**Hypermedia As the Engine of Application State (HATEOAS)**

1. **What is HATEOAS**

* **HATEOAS** is a constraint of the REST application architecture
* Keeps the RESTful style architecture **unique from most other network application** architectures
* Uses **hypermedia** to describe what future actions are available to the client
* Allowable actions are derived in the API based on the current application state and returned to the client as a **collection of links**
* Client uses these **links to drive further** interactions with the API
* Tells the client what **options** are **available** at a given point in time.
* Doesn't tell them how each link should be used or exactly what information should be sent
* It is conceptually the same as a **web user browsing** through web pages by clicking the **relevant hyperlinks**to achieve a final goal

**HATEOAS Example**

* Simple response **without** using **HATEOAS**
* We have a simple REST controller that returns entity in JSON format to the client

{ "id" :2, "name": "Pesho", "age":12 }

**Using** **HATEOAS**

{ "id":2,

"name":"Pesho",

"age":12,  
 "\_links":{

"self":{"href":"http://localhost:8080/students/2"},

"delete":{"href":"http://localhost:8080/students/delete/2"},

"update":{"href":"http://localhost:8080/students/update/2"},

"orders":{"href":"http://localhost:8080/orders/allByStudentId/2"}

}

}

**Rel & Href**

* **rel** - describes the **relationship** between the Student resource and the URL
  + In example above – self, update, delete …
  + **describes** the **action** that's performed with the link
  + It's important that this **value** is intuitive as it **describes** the purpose of the link
* **href** - the **URL** used to perform the action described in rel

1. **Implement HATEOAS in Spring**

* Adding hypermedia links to RESTful responses is something you could implement on your own, but …
* **Spring HATEOAS** makes it **very easy**

<dependency>

<groupId>org.springframework.hateoas</groupId>

<artifactId>spring-hateoas</artifactId>

<version>1.1.0.RELEASE</version>

</dependency>

1. **Example App DB**

* Our example app have small base with some relations between entities
  + We have Students, Orders and Courses

**Prepare Controllers to Work**

* If we implementing **RepresentationModel <T>** we can added links directly to our entity
* We need two methods from **WebMvcLinkBuilder**, that’s why we must import them

import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;

import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;

**Main Work in Controller**

... // Without implementing RepresentationModel<T>

Optional<Student> studentOpt = this.studentRepository.findById(id);

return studentOpt  
 .map(s -> ResponseEntity.ok(  
 EntityModel.of(s, getStudentLinks(s))))  
 .orElse(ResponseEntity.notFound().build());

}

private Link[] getStudentLinks(Student student) {

Link self = linkTo(methodOn(StudentsController.class)

.getStudent(student.getId()))

.withSelfRel();

Link orders = linkTo(methodOn(StudentsController.class)

.getAllOrdersByStudentId(student.getId()))

.withRel("orders");

return List.of(self, orders).toArray(new Link[0]);

}

**... // Implementing RepresentationModel<T>**

Student student = this.studentService.findById(id);

student.add(linkTo(methodOn(StudentsController.class)

.getStudent(student.getId()))  
 .withSelfRel());

student.add(linkTo(methodOn(StudentsController.class)

.deleteStudent(student.getId()))  
 .withRel("delete"));

student.add(linkTo(methodOn(StudentsController.class)

.updateStudent(student.getId(), student))  
 .withRel("update"));

student.add(linkTo(methodOn(OrdersController.class)

.findAllOrdersByUserId(student.getId()))

.withRel("orders"));

return ResponseEntity.ok(student);

1. **Benefits of Using HATEOAS**

* **URL structure** of the API can be **changed without affecting** clients
* If the URL structure is changed in the service, clients will automatically pick up the new URL structure   
  via hypermedia
* Hypermedia APIs are **explorable**
* Guiding clients toward the next step in the workflow by **providing** only the **links** that are **relevant** based on the current application state

1. **Negatives of Using HATEOAS**

* Adds **extra complexity** to the API, which affects to:
  1. **developer** needs to handle the **extra work** of adding links to each response
  2. **more complex** to **build** and **test** than a vanilla CRUD REST API
  3. **clients** also have to deal with the **extra complexity** of **hypermedia**

1. **HAL Explorer**
   * To use **HAL Explorer** we need to add **dependency**

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-explorer</artifactId>

</dependency>

**Error Handling**

1. **What is error handling?**

* **Error handling** refers to:
  + The **anticipation**, **detection** and **resolution** of programming errors
  + The response & recovery procedures from error conditions
* Error handling is necessary!
  + Improves **user experience**
  + **Optimizes** debugging
  + Facilitates **code maintenance**
  + Ensures **product quality**

1. **Error Handling in Spring**

* Spring MVC offers **no default** (fall-back) **error page** out of the box, however Spring Boot does
* At start-up, Spring Boot **tries to find** a mapping for **/error**
* Spring MVC provides **several approaches** to error handling
  + Per exception
  + Per controller
  + Globally
* Each option has its own use cases and circumstances
* You can use:
  + **Response-annotated** custom exceptions
  + **Controller-based** handlers on specified actions
  + **@ControllerAdvise** annotated classes for global handlers

1. **Custom error page**

* To disable the default **White label error page**   
  for a Spring Boot application:
  + We must save **error.html** file in resources/templates directory, it'll automatically be picked up by the default Spring

1. **ErrorController Interface**

* Spring Boot maps **/error** to **BasicErrorController** which populates model with error attributes   
  and then returns 'error' as the view name
* To replace BasicErrorController with our own custom controller which can map to /error, we need to **implement** **ErrorController** interface

@Controller

public class MyErrorController implements ErrorController {

@RequestMapping

@ResponseBody

public String handle(HttpServletRequest request){

//Some code ...

}

}

1. **HTTP Status Codes**

* Unhandled exceptions during a request produce HTTP 500 response
* Any custom exception can be annotated with **@ResponseStatus**
  + Supports all HTTP status codes
  + When thrown and unhandled – produces error page with   
    appropriate response

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "Product was not found.")

public class ProductNotFoundException extends RuntimeException {

*// Exception definition*

}

* And the controller action, throwing the exception

@GetMapping("/products/details/{id}")

public ModelAndView productDetails(@PathVariable String id, ModelAndView modelAndView) {

Product product = this.productRepository.findProductById(id);

if(product == null) throw new ProductNotFoundException(id);

modelAndView.addObject("product", product);

return this.view("product/details", modelAndView);

}

1. **Controller-Based Error Handling**

* You can define Controller-specific Exception Handlers
  + Annotated with **@ExceptionHandler** annotation
  + They work **only** for the **Controller** they are defined in
  + Can be annotated with **@ResponseStatus** to convert HTTP status
  + Can accept the **caught** **exception** as a **parameter**
  + Can return **ModelAndView** or **String** (view name)
  + Can catch **multiple** exception types

@ExceptionHandler({PersistenceException.class, TransactionException.class})

public ModelAndView handleDbExceptions(DatabaseException e) {

ModelAndView modelAndView = new ModelAndView("error");

modelAndView.addObject("message", e.getMessage());

return modelAndView;

}

View-------------------------------------------------------------------------------------------

<html>

<head>...</head>

<body>

<h1>An error occurred while processing your request!</h1>

<p th:text="|Error: ${message}|"></p>

</body>  
</html>

* Handler methods have **flexible signatures**
  + You can pass in servlet-related objects as parameters
    - **HttpServletRequest**
    - **HttpServletResponse**
    - **HttpSession**
    - **Principal**
* The **Model** or **ModelAndView** cannot be a parameter though
  + Instead of passing it, you have to setup it inside the method
  + Nevertheless, this is not an issue because the **IoC** **container** would have done the same (pass an **empty instance**)
* It is not a good practice for full error **stacktraces** to be exposed
  + Your users don't need to see ugly exception web-pages
  + You may even have security policies which **strictly forbid** any public exception info
  + Hide as **much information** as **possible** and present **User-friendly** error pages
  + For **testing** purposes you may view details
    - This may need an **environment** setup

1. **Global Application Exception Handling**

* There is a way to achieve Global exception handling in Spring
  + This is done through the **@ControllerAdvise** annotation
* Any class annotated with **@ControllerAdvise** turns into an **interceptor-like** controller:
  + Enables **global exception handling**
  + Enables **model enhancement** methods
* In **@ControllerAdvice** classes you still use **@ExceptionHandler**
  + However, this time it refers to the whole application
  + The error handling is not limited only to a specific controller

**@ControllerAdvice**

**public class GlobalExceptionHandler {**

**@ExceptionHandler({TransactionException.class, PersistenceException.class})**

**public ModelAndView handleDatabaseErrors(DatabaseException e) {**

**ModelAndView modelAndView = new ModelAndView("index");**

**modelAndView.addObject("message", e.getMessage());**

**modelAndView.addObject("stack", {...} */\* Formatted Stack Trace \*/*);**

**return modelAndView;**

**}}**

1. **Global Exception Handling (REST)**

* **RESTful requests** may also generate unexpected exceptions
  + HTTP Error response codes are a good choice
  + However sometimes you might need more than just a status
    - **Customized Error Object**, which can be presented on the Client
    - **Limited Information** returned to the Client
* You can customize the **Error Response** by introducing a class
  + The **Error Handler** itself remains the same as in casual web apps

public class ErrorInfo {

public final String url;

public final String ex;

public ErrorInfo(String url, Exception ex) {

this.url = url;

this.ex = ex.getLocalizedMessage();

}

}

@ControllerAdvice

public class GlobalRESTExceptionHandler {  
 @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

@ExceptionHandler({TransactionException.class,

PersistenceException.class})

public @ResponseBody ErrorInfo handleRESTErrors(HttpServletRequest req,

DbException e) {

return new ErrorInfo(req.getRequestURL(), ex);

}

}

1. **Exception Techniques Use Cases**

**What to Use When?**

* Spring offers **many** choices, when it comes to **error** handling
* Be **careful** **mixing** too **many** of these
  + You may not get the behavior you wanted
* There are some semantics, that should be followed, though
  + For custom exceptions, consider adding **@ResponseStatus** to then
  + For Controller-specific exceptions, **@ExceptionHandler** methods should be added alongside the actions
  + For all other exceptions, **@ExceptionHandler** methods in **@ControllerAdvice** classes should be implemented