DESARROLLO DE APLICACIONES JEE CON SPRING FRAMEWORK

Dirección Nacional de Educación Continua

13-11-2021 Módulo 4

INTRODUCCIÓN

- La presente tiene como objetivo explicar las capacidades de un entorno de trabajo, sus funciones y capacidades, además explicar en detalle algunos de los módulos del framework de spring, con la intención de preparar al lector para generar soluciones empresariales simples y efectivas, conociendo la arquitectura y con esto ser un aporte al momento de diseñar una aplicación, aplicando los conocimientos de este documento.

RESUMEN

- En el Módulo anterior, hemos revisado lo siguiente:
- Datasource en Spring.
- Creando un datasource con jdbcTemplate
- Acceso a datos mediante JPA
- ¿Que es la transaccionalidad?
- Interacción con las entidades
- Clase Repositorio

CONTROL DE ACCESO MEDIANTE SPRING SECURITY

- Spring Security es un framework de apoyo al marco de trabajo Spring, que dota al mismo de una serie servicios de seguridad aplicables para sistemas basados en la arquitectura basados en JEE, enfocado particularmente sobre proyectos construidos usando SpringFramework. De esta dependencia, se minimiza la curva de aprendizaje si ya es conocido Spring.
- Spring Security tiene un alto grado de aceptación entre la comunidad de desarrolladores por su gran flexibilidad sobre los modelos de autenticación. De esta manera, permite una rápida integración de las soluciones que presentan ante los requerimientos de los clientes potenciales, sin implicar una migración de los sistemas a un determinado entorno. Además, es una plataforma abierta y en constante evolución, lo que permite ir añadiendo nuevos mecanismos de autenticación, o directamente programar los propios de manera poco compleja

INCORPORANDO EL MÓDULO SPRING SECURITY AL PROYECTO

```java
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        prePostEnabled = true,
        securedEnabled = true,
        jsr250Enabled = true)

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    // Create 2 users for demo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user").password("{noop}password").roles("USER")
                .and()
                .withUser("admin").password("{noop}password").roles("USER", "ADMIN");

    }
```

```xml
<!-- spring security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- Para incorporar el módulo de seguridad al proyecto es necesario modificar el POM:xml
- Además la creación de un filtro de configuración

CONFIGURACIÓN BÁSICA DE SPRING SECURITY

- Existen dos formas de configurar Spring Security: una 100% manual y otra algo más automática. La forma manual consiste en definir todos los filtros necesarios para su correcto funcionamiento.
  La otra forma, que es la que mostraremos, consiste en usar el elemento http. Este elemento configurará por nosotros todos los filtros obligatorios, beans y la cadena de seguridad. Si además se le pone el atributo auto-config a true, configurará todos los filtros restantes necesarios para el funcionamiento.

- El primero de los beans que se crea es el springSecurityFilterChain, que es el encargado de gestionar los diferentes filtros de la cadena de seguridad.

- Obviamente, se pueden personalizar los filtros que se configuran automáticamente e incluso añadir los nuestros propios. Esto se realiza anidando un nuevo elemento en el http. Con el custom-filter, se pueden añadir nuevos filtros, así como posicionarlos donde sea necesario en la cadena de seguridad. Es posible que alguno de los nuevos filtros entren en conflicto con algunos de los existentes. Esto ocurrirá si estamos sobreescribiendo un filtro existente, con lo cual, habrá que poner el atributo auto-config a false..

CREANDO UN FORMULARIO DE LOGIN

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        prePostEnabled = true,
        securedEnabled = true,
        jsr250Enabled = true)

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    // Create 2 users for demo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user").password("{noop}password").roles("USER")
                .and()
                .withUser("admin").password("{noop}password").roles("USER", "ADMIN");

    }
```

```
<!DOCTYPE HTML>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <title>Spring Boot Thymeleaf Hello World Example</title>

    <link rel="stylesheet" th:href="@{webjars/bootstrap/4.2.1/css/bootstrap.min.css}"/>
    <link rel="stylesheet" th:href="@{/css/main.css}"/>

</head>

<body>

<nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
    <a class="navbar-brand" href="#">AIEP</a>
    <h3><span sec:authentication="name"></span> | <a style="color: white;" onclick="document.forms['logoutForm'].submit()">Logout</a> | </h3>
</nav>

<main role="main" class="container">
    <form id="logoutForm" method="POST" th:action="@{/logout}">
            <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
        </form>
    <div class="starter-template">
        <h1>Spring Boot Web Thymeleaf Example</h1>
        <h2>
            <span th:text="'Hello, ' + ${message}"></span>
        </h2>

    </div>


    <ol>
        <li th:each="task : ${tasks}" th:text="${task}"></li>
    </ol>

</main>
<!-- /.container -->

<script type="text/javascript" th:src="@{webjars/bootstrap/4.2.1/js/bootstrap.min.js}"></script>
</body>
</html>
```

- Para la creación de un formulario Login primero crearemos una clase filtro de seguridad.
- Luego utilizaremos, lo aprendido en el módulo anterior de Thymeleaf, agregando las configuraciones asociadas quedando de la siguiente forma
- En este fichero Welcome.html podemos ver el código asociado al mensaje de bienvenida y el código de Logout

CREANDO UN FORMULARIO DE LOGIN

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
<meta name="description" content="">
<meta name="author" content="">
<title>Spring Security Example</title>
<link href="/webjars/bootstrap/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h2 class="form-signin-heading">Welcome to AIEP, please login</h2>
        <div th:if="${param.error}" class="alert alert-danger">
            Invalid username and password.
        </div>
        <div th:if="${param.logout}" class="alert alert-success">
            You have been logged out.
        </div>
        <form class="form-signin" method="POST" th:action="@{/login}">
            <p>
                <label for="username" class="sr-only">Username</label>
                <input type="text" id="username" name="username" class="form-control"
                    placeholder="Username" required autofocus>
            </p>
            <p>
                <label for="password" class="sr-only">Password</label>
                <input type="password" id="password" name="password" class="form-control"
                    placeholder="Password" required>
            </p>
            <button class="btn btn-lg btn-primary btn-block" type="submit">Login</button>
        </form>
    </div>
</body>
</html>
```

- El fichero Login.html se ve de la siguiente forma.

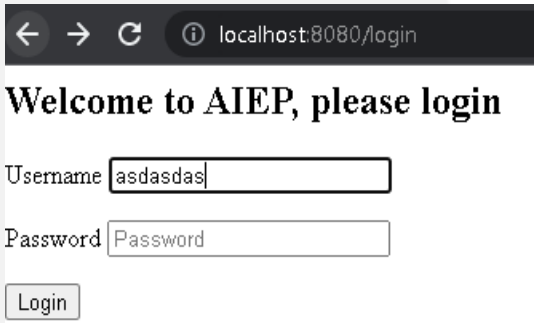
En él podemos validar el Html asociado a el ingreso de usuario y password

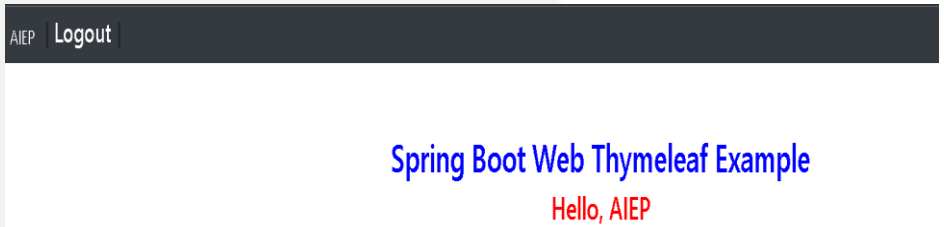
REALIZANDO UN LOGIN Y LOGOUT DE UNA APPLICACIÓN

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
            //HTTP Basic authentication
            .httpBasic()
            .and()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/books/**").hasRole("USER")
            .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
            .antMatchers(HttpMethod.PUT, "/books/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.PATCH, "/books/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE, "/books/**").hasRole("ADMIN")
            .and()
            .csrf().disable()
            .formLogin().loginPage("/login")
            .permitAll()
            .and()
            .logout()
            .permitAll();
}
```

- Para el caso de un Login y Logout debemos crear una clase filtro de Spring Security
- En la imagen podemos ver como se configura el Login y su url , además la configuración del logout
- Una vez realizado el cambio, procedemos a probar el funcionamiento de la app construida con STS

REALIZANDO UN LOGIN Y LOGOUT DE UNA APPLICACIÓN





- Una vez realizado las configuraciones se verá de la siguiente forma.

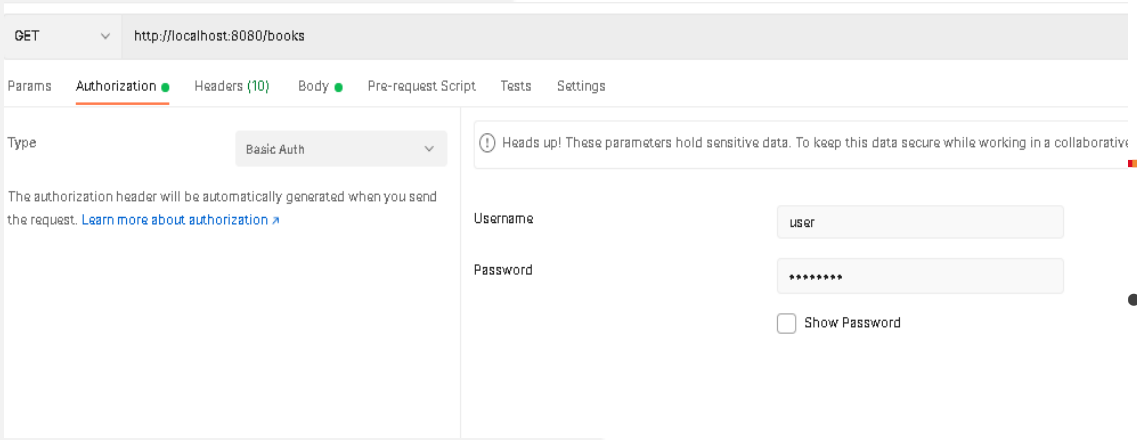
MANEJO DE ROLES

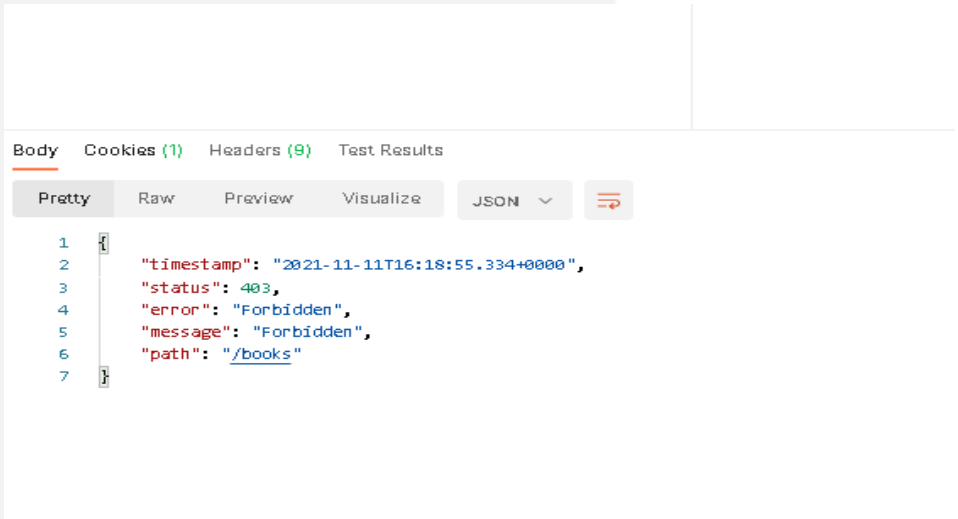
```java
@Configuration
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    // Create 2 users for demo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication()
                .withUser("user").password("{noop}password").roles("USER")
                .and()
                .withUser("admin").password("{noop}password").roles("USER", "ADMIN");

    }
```

```java
// Secure the endpoins with HTTP Basic authentication
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
            //HTTP Basic authentication
            .httpBasic()
            .and()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/books/**").hasRole("USER")
            .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
            .antMatchers(HttpMethod.PUT, "/books/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.PATCH, "/books/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE, "/books/**").hasRole("ADMIN")
            .and()
            .csrf().disable()
            .formLogin().disable();
}
```

- Como vimos en el ejemplo anterior, pudimos crear 2 roles en memoria para autenticarnos.
- Como se ve en la imagen (abajo), en el caso de los métodos que obtienen información cualquier usuario puede consultarlos, y los que administran solo el Admin.

AÑADIENDO SEGURIDAD A LOS ELEMENTOS DE LA CAPA VISTA



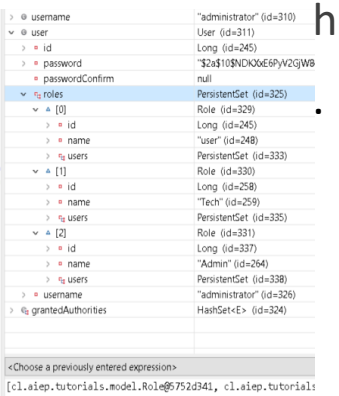


- De acuerdo con lo revisado anteriormente los métodos ya se encuentran segurizados. Para probarlos utilizaremos una herramienta libre llamada PostMan, en el primer método GET puede ser consultado por un usuario como la siguiente imagen
- En el caso que no se ingrese la autenticación básica, el servicio rechaza la transacción

OBTENIENDO USUARIO AUTENTICADO EN EL CONTROLADOR

```
    }

    @PostMapping("/registration")
    public String registration(@ModelAttribute("userForm") User userForm, BindingResult bindingResult) {
        userValidator.validate(userForm, bindingResult);

        if (bindingResult.hasErrors()) {
            return "registration";
        }

        userService.save(userForm);

        securityService.autoLogin(userForm.getUsername(), userForm.getPasswordConfirm());

        return "redirect:/welcome";
    }

    @GetMapping("/login")
    public String login(Model model, String error, String logout) {
        if (securityService.isAuthenticated()) {
            return "redirect:/";
        }

        if (error != null)
            model.addAttribute("error", "Your username and password is invalid.");

        if (logout != null)
            model.addAttribute("message", "You have been logged out successfully.");

        return "login";
    }
```

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) throw new UsernameNotFoundException(username);

        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        for (Role role : user.getRoles()){
            grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
        }

        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), grantedAuth
    }
}
```



- Una vez autenticado el usuario, es factible captura sus roles, para esto construiremos un service que permita obtener los roles desde BD y este sea invocado desde el Controller, en este caso utilizaremos la herramienta para debug de STS

SPRING Framework

```
@GetMapping("/")
public String defaultAfterLogin(Model model) {

    String retorno="welcome";
     model.addAttribute("message", "AIEP");
  return retorno;
}
```

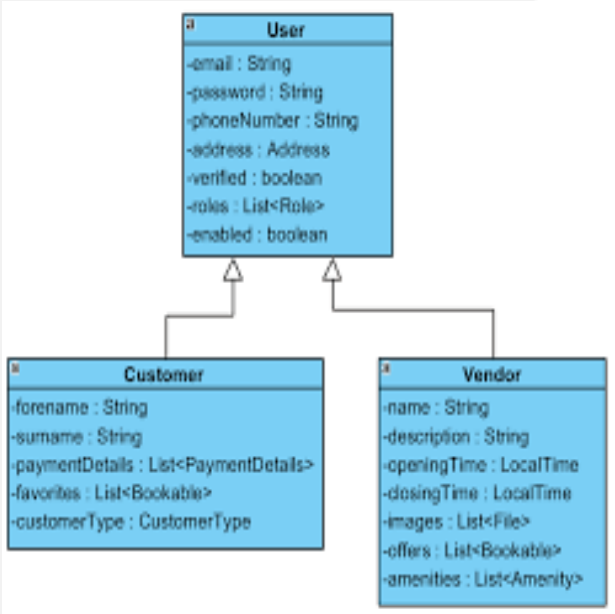
```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        prePostEnabled = true,
        securedEnabled = true,
        jsr250Enabled = true)

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
```

```
@GetMapping("/")
@PreAuthorize("isAuthenticated()")
public String defaultAfterLogin(Model model) {

    String retorno="welcome";
     model.addAttribute("message", "AIEP");
  return retorno;
}

@Autowired
```

AGREGANDO SEGURIDAD EN LOS CONTROLADORES MEDIANTE ANOTACIONES

- Para agregar seguridad en los controladores, utilizaremos la anotación @PreAuthorize, anotación que permite evitar los accesos no validados a los métodos del controlador.
- En este caso el controlador no tiene ningún control sobre la autenticación del usuario, por ende es vulnerable.
- En cambio con la anotación ya no es posible acceder al método sin estar autenticado
- Es importante incluir dentro del filtro de seguridad que permita la anotación para autenticar.

AUTENTICACIÓN CONTRA UNA BD



- La autenticación contra una BD indica que los usuarios sus password y atributos serán almacenadas en Tablas para luego ser consultadas.
- Este método no es recomendable desde el punto de vista seguridad y de reutilización ya que a pesar de que es factible de realizar, en la actualidad existen herramientas para la administración de cuentas de usuario y sus contraseñas como los LDAP, ActiveDirectory, etc. Que separan los usuarios y sus cuentas de un negocio especifico o modelo.

AUTENTICACIÓN UTILIZANDO JPA

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) throw new UsernameNotFoundException(username);

        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        for (Role role : user.getRoles()){
            grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
        }

        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), grantedAuthorities);
    }
}
```

- Complementando lo previamente indicado en el documento la autenticación contra Database se puede ejecutar con jdbcTemplate o Entidades, para el ejemplo a continuación utilizaremos Entidades.
- Lo primero que debemos modificar el archivo POM.xml, en este caso se debe incluir la BD.
- Una vez realizado esto, se deben construir las entidades que soporten el modelo asociado a usuarios. En este caso son las siguientes clases
- 2 entidades, User y Role como se explica en el módulo anterior las entidades son una representación de las tablas en BD
- 2 services usuarioService y RoleService, los services son capas de abstracción para la capa de acceso a datos
- 2 Repositorios UserRepo, RoleRepo, los repositorios son clases que permiten realizar las consultas a Base de Datos
- 1 Controller que invoca al service, Clase en donde se encuentran las Urls de la aplicación y se realizan las validaciones de formulario.

PRUEBA PRÁCTICA NRO4

- El alumno deberá crear un proyecto starter spring boot (reutilizar Práctica 2)
- Configurar el proyecto para thymeleaf (crear 1 htmls de Login y y modificar el Welcome (logout))
- Crear una clase filtro con los usuarios en memoria y los atributos para segurizar las APIS.
- Modificar una clase RestController con 2 métodos
  - Método Login –> si es correcto devuelve HTML welcome
  - Método Base sin URL “/”
- Segurizar todos los métodos del proyecto anterior asegurándose que esté autenticado para acceder, con excepción del /Login.
- Incluir las librerías necesarias en el POM.xml para ejecutar las pruebas

AIEP
de la Universidad
Andrés Bello

Certified
B
Corporation

MUCHAS GRACIAS