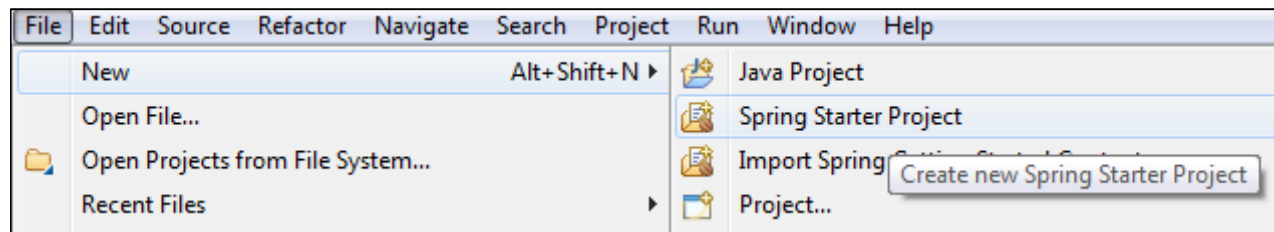


CREACION DE UN SERVICIO API REST MONGO SENCILLO CON JWT

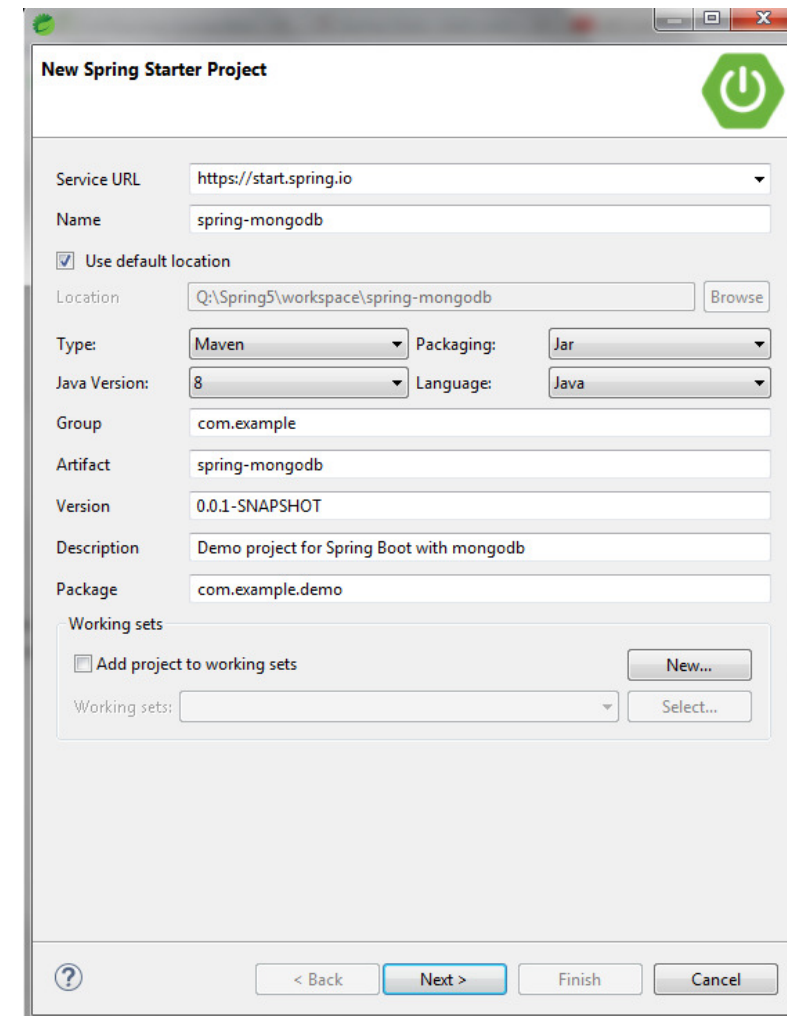
1. Proyecto Mongo Spring
2. Clases Entity-Repository
3. Controlador Rest
4. JWT (JSON WEB TOKEN)

1. PROYECTO MONGO SPRING

Paso 1) Creamos un proyecto Spring Boot, en la opción de menu File/New/Spring Starter Project:



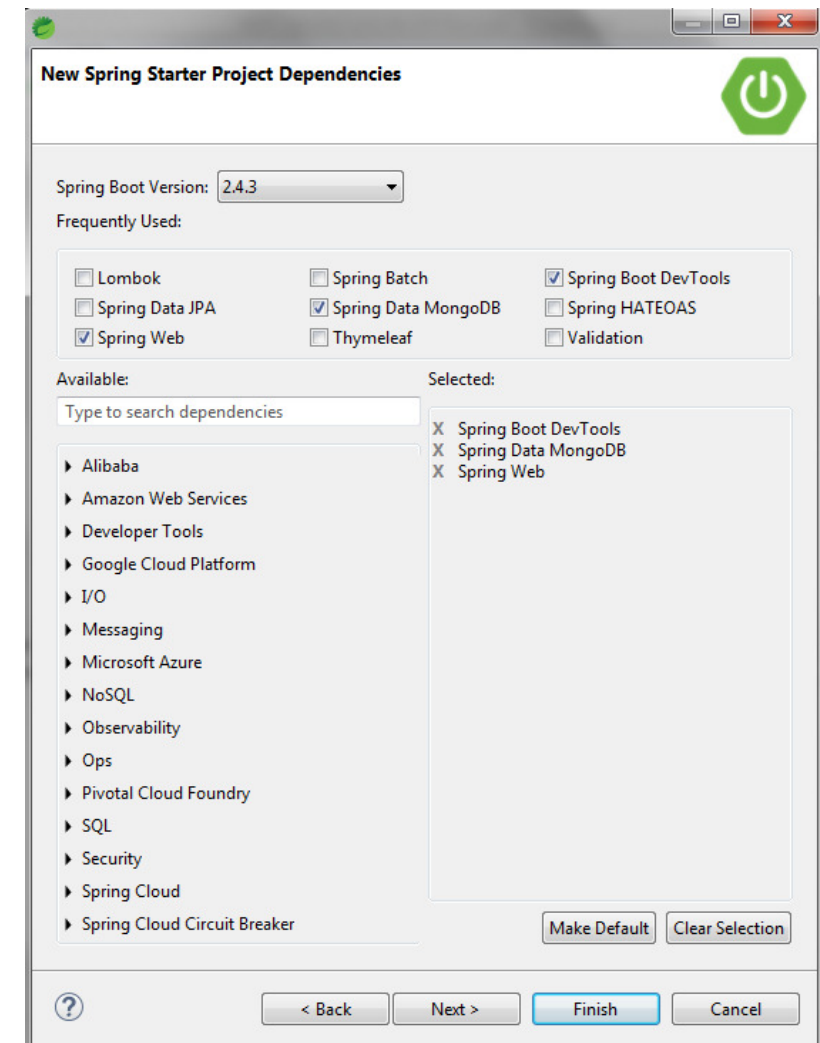
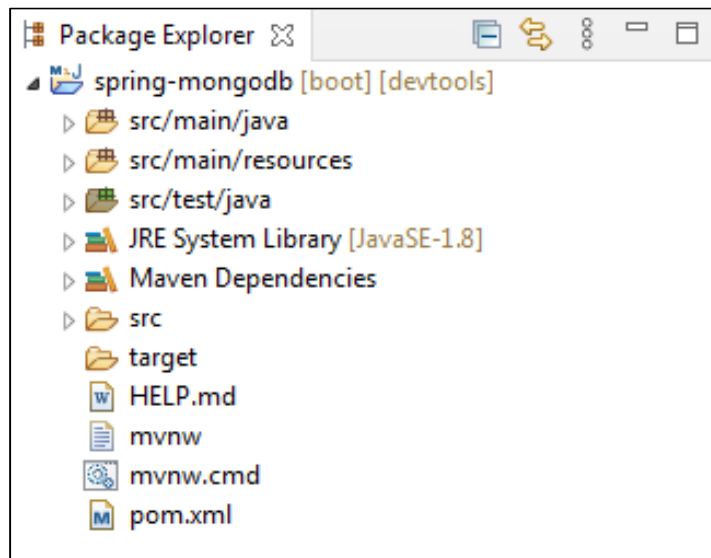
Podemos dejar por defecto los valores que nos presenta el wizard. Si se desea se puede cambiar el nombre de proyecto, el package raíz, el tipo de proyecto (Maven o Gradle) y/o la versión de Java.



1. PROYECTO MONGO SPRING

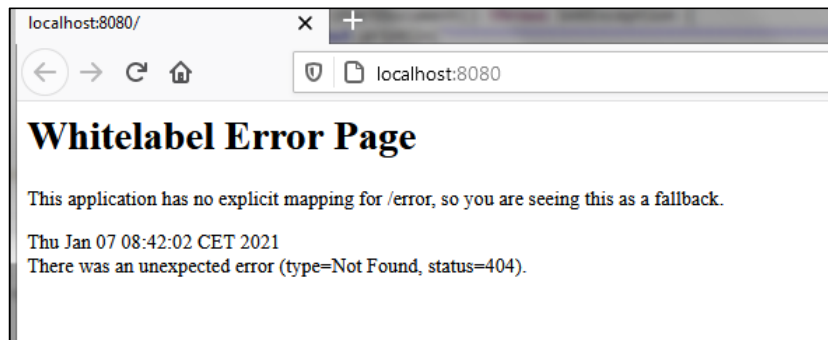
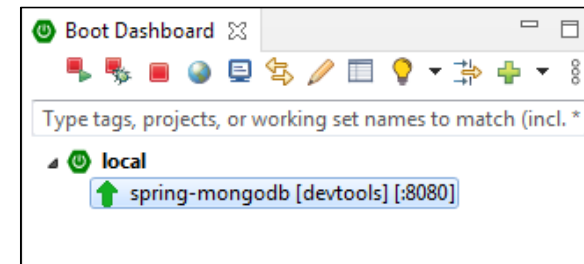
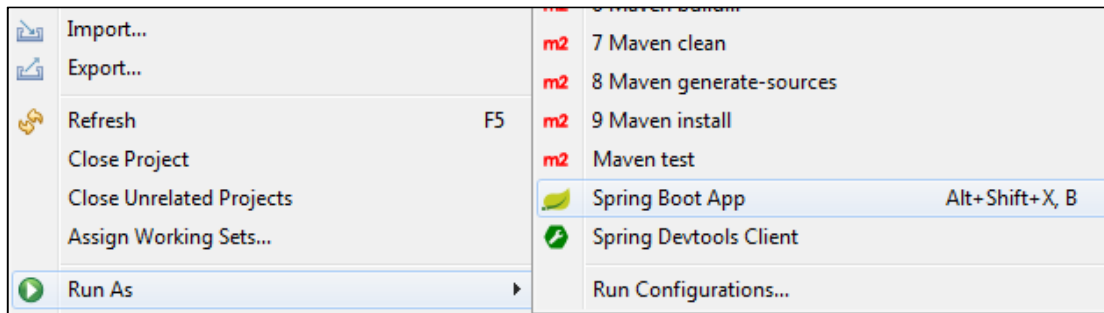
Paso 2) Agregamos las librerías:

- Spring Web (imprescindible)
- Spring Data MongoDB (imprescindible)
- Spring Boot Dev Tools (no imprescindible)



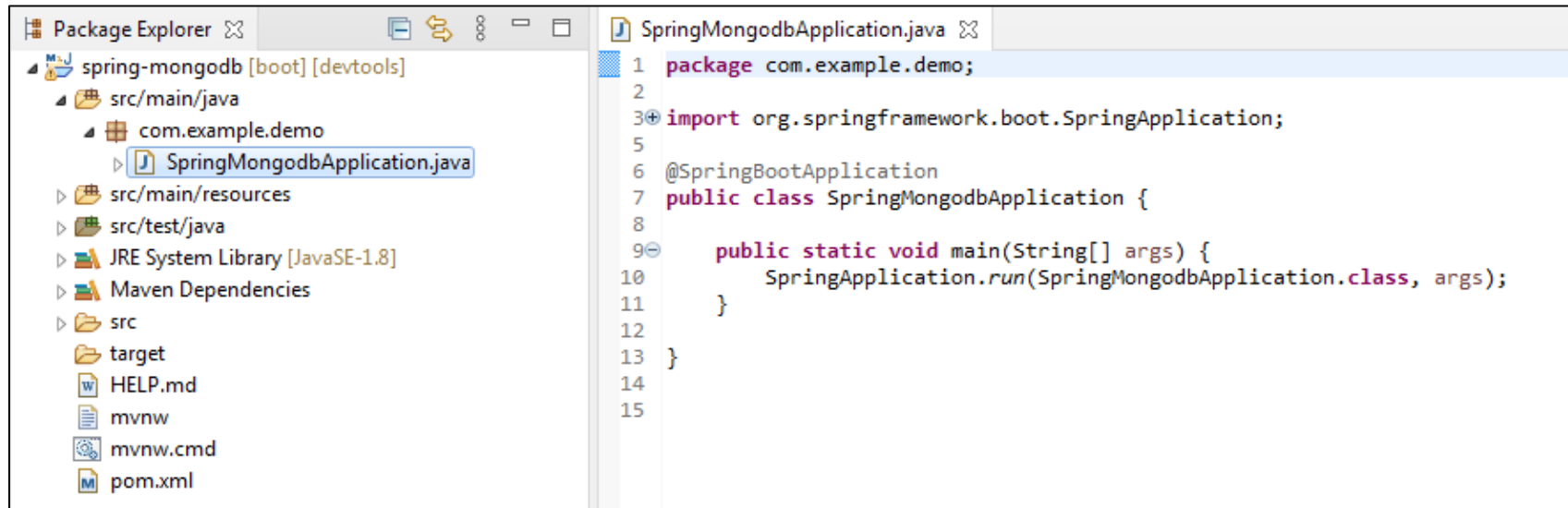
1. PROYECTO MONGO SPRING

Paso 3) Probamos de ejecutar el proyecto, para ello levantamos el servidor Tomcat haciendo Run As/Spring Boot App. Una vez iniciado el servidor, probamos **localhost:8080** en un navegador. Nos da error porque no tenemos ninguna página de inicio. Por otro lado indica que hay un servidor respondiendo en el puerto 8080.



1. PROYECTO MONGO SPRING

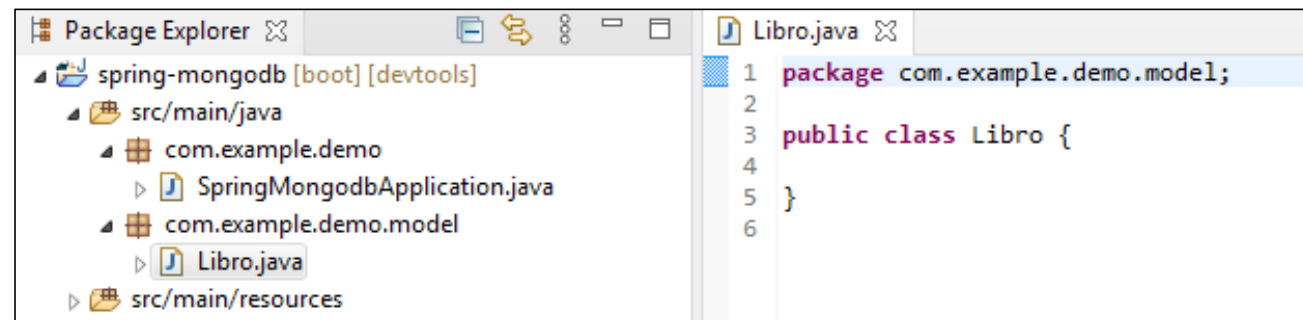
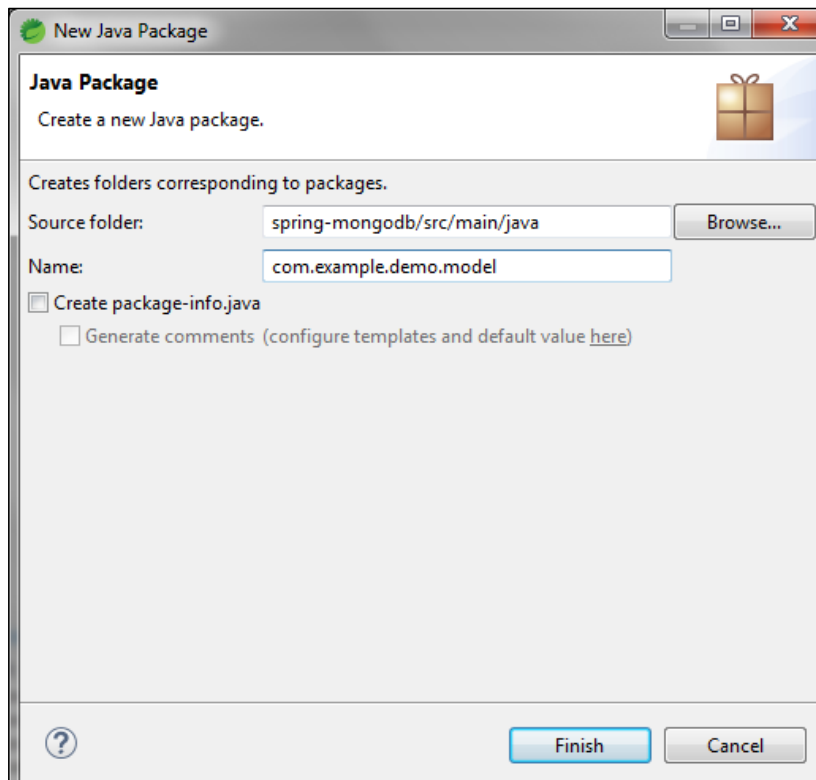
Paso 4) Podemos observar en el package raíz indicado al principio en la creación del proyecto, la clase generada automáticamente que inicia nuestro servidor y la aplicación:



```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringMongodbApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringMongodbApplication.class, args);
10    }
11 }
12
13
14
15
```

2. CLASES ENTITY-REPOSITORY

Paso 1) Creamos la clase Libro dentro del nuevo package model:



2. CLASES ENTITY-REPOSITORY

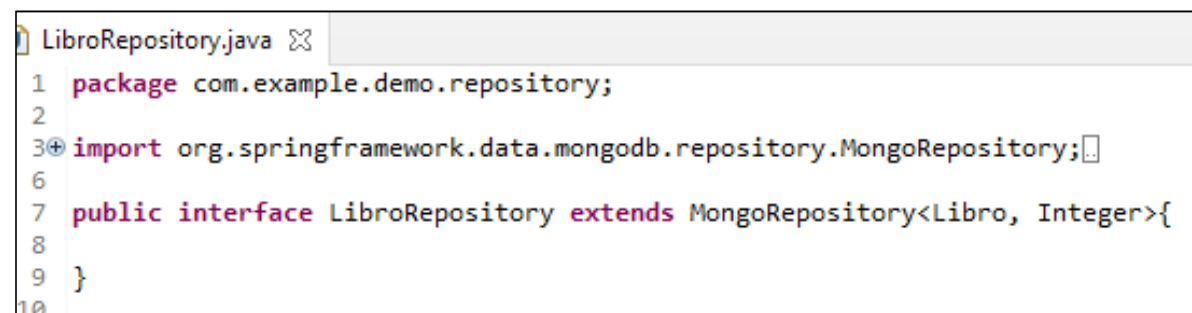
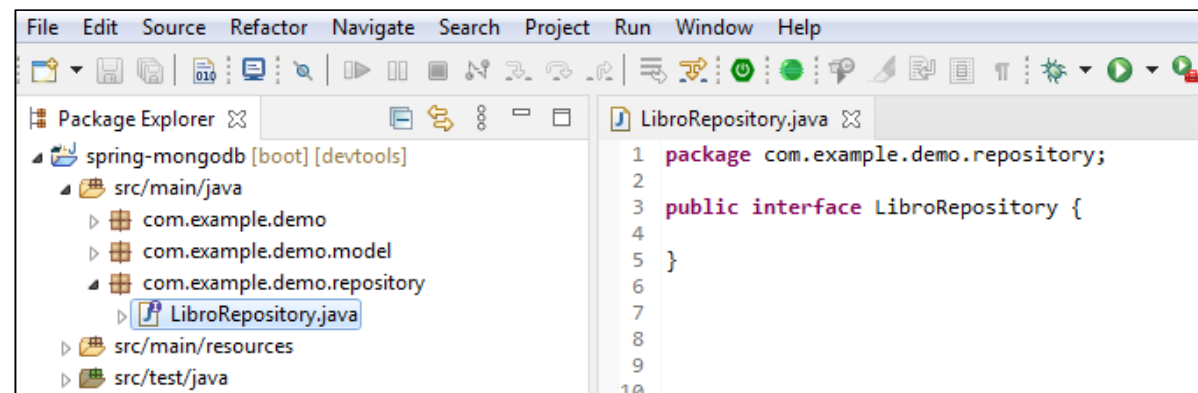
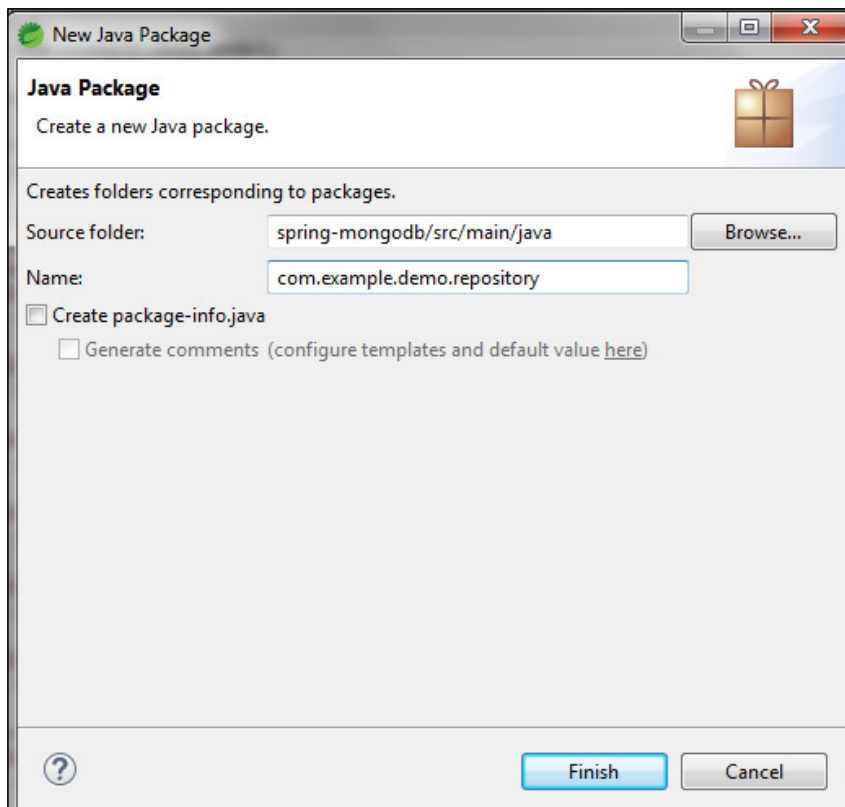
Paso 2) Creamos 4 atributos en la clase Libro, junto con sus getters y setters y la clase toString:

Mediante la anotación
@Document indicamos la
colección que representa esta
clase dentro de la base de
datos mongo

```
Libro.java x
1 package com.example.demo.model;
2
3 import org.springframework.data.annotation.Id;
4
5
6
7 @Document(collection = "libros")
8 public class Libro {
9     @Id
10    private int id;
11    @Field (name = "nombre")
12    private String nombre;
13    @Field (name = "autor")
14    private String autor;
15    @Field (name = "editorial")
16    private String editorial;
17
18    public Libro() {
19    }
20
21    @Override
22    public String toString() {
23        return "Libro [id=" + id + ", nombre=" + nombre +
24            ", autor=" + autor + ", editorial=" + editorial + "];"
25    }
```

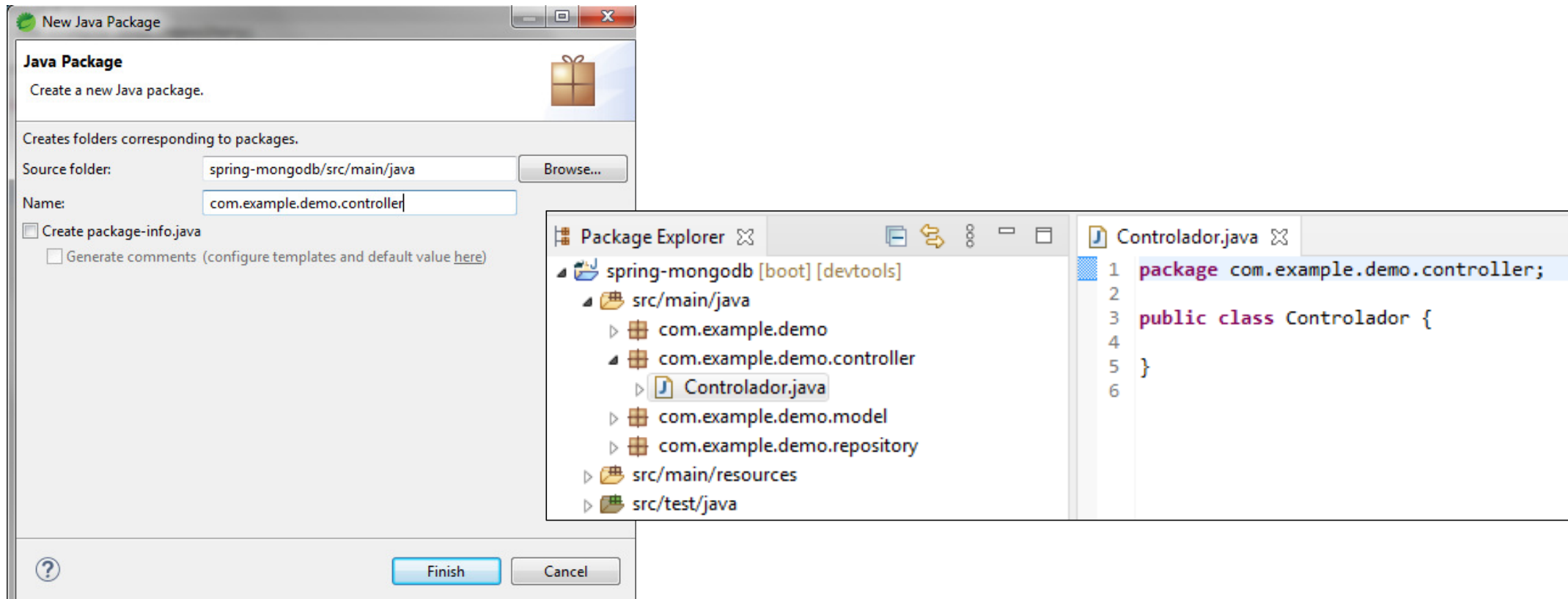

2. CLASES ENTITY-REPOSITORY

Paso 3) Creamos LibroRepository que será nuestra clase DAO, dentro de su correspondiente package. Deriva de MongoRepository



3. CONTROLADOR REST

Paso 1) Creamos el package controller y dentro creamos el controlador API REST.



The screenshot shows the Eclipse IDE interface. On the left, the 'New Java Package' dialog is open. It has a title bar 'New Java Package' and a subtitle 'Java Package'. Below the subtitle, it says 'Create a new Java package.' and 'Creates folders corresponding to packages.' The 'Source folder:' field is set to 'spring-mongodb/src/main/java' with a 'Browse...' button. The 'Name:' field is set to 'com.example.demo.controller'. There are checkboxes for 'Create package-info.java' (checked) and 'Generate comments' (unchecked). At the bottom are 'Finish' and 'Cancel' buttons. On the right, the 'Package Explorer' shows the project structure: 'spring-mongodb [boot] [devtools]' contains 'src/main/java', which contains 'com.example.demo', which contains 'com.example.demo.controller'. Under 'com.example.demo.controller', 'Controlador.java' is listed. To the right of the Package Explorer, the 'Controlador.java' file is open, showing the following code:

```
1 package com.example.demo.controller;
2
3 public class Controlador {
4
5 }
6
```

3. CONTROLADOR REST

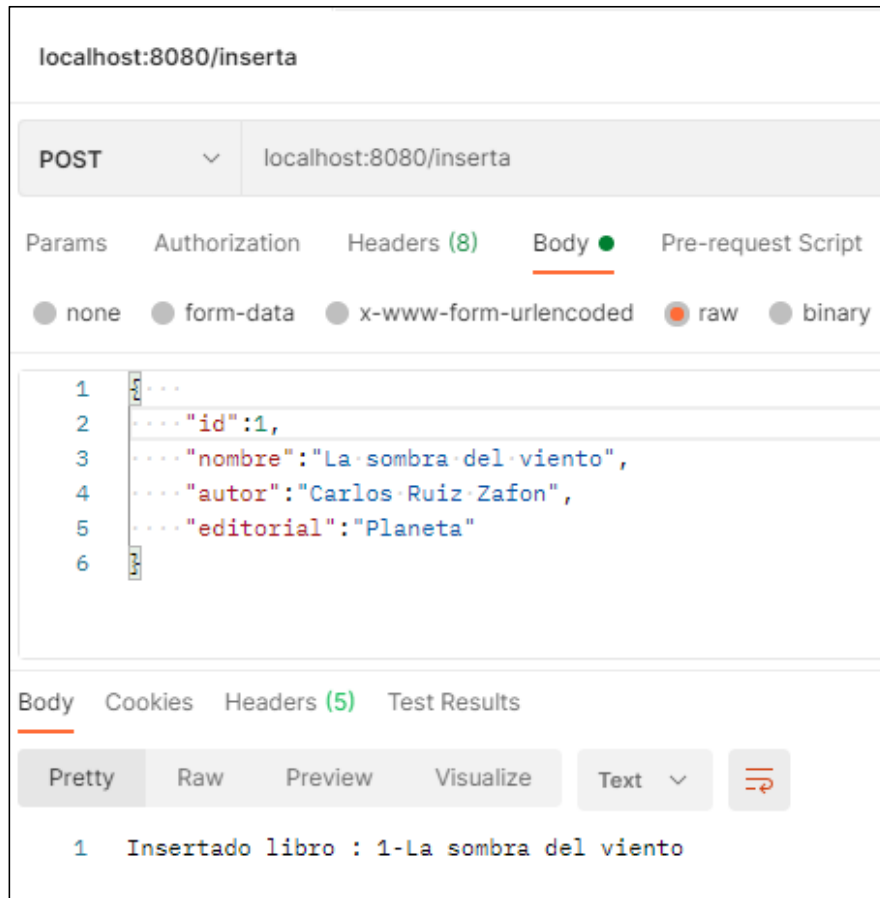
Paso 2) Ponemos la etiqueta `@RestController` al controlador. Inyectamos `LibroRepository` y creamos dos primeros servicios rest:

- Inserta → save
- getBooks → findAll

```
Controlador.java
14 @RestController
15 public class Controlador {
16
17     @Autowired
18     private LibroRepository repositorio;
19
20     @PostMapping("/inserta") //localhost:8080/inserta
21     public String saveBook(@RequestBody Libro libro) {
22         repositorio.save(libro);
23         return "Insertado libro : " + libro.getId()+"-"+libro.getNombre();
24     }
25
26     @GetMapping("/") //localhost:8080/
27     public List<Libro> getBooks() {
28         List<Libro> lista= repositorio.findAll();
29         return lista;
30     }
31 }
```


3. CONTROLADOR REST

Paso 4) Probamos el servicio insertar con Postman y comprobamos el resultado de la inserción mediante el programa MongoDBCompass:



localhost:8080/inserta

POST localhost:8080/inserta

Params Authorization Headers (8) Body Pre-request Script

none form-data x-www-form-urlencoded raw binary

```
1 { ...
2   ... "id": 1,
3   ... "nombre": "La sombra del viento",
4   ... "autor": "Carlos Ruiz Zafon",
5   ... "editorial": "Planeta"
6 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text

1 Insertado libro : 1-La sombra del viento



biblioteca.libros

Documents Aggregations Schema

FILTER { field: 'value' }

ADD DATA VIEW

```
_id: 1
nombre: "La sombra del viento"
autor: "Carlos Ruiz Zafon"
editorial: "Planeta"
_class: "com.example.demo.model.Libro"
```

3. CONTROLADOR REST

Paso 5) Nuevamente insertamos un segundo documento, pero ahora comprobamos el resultado desde Postman llamando al handler getBooks:

localhost:8080/inserta

POST localhost:8080/inserta

Params Authorization Headers (8) **Body** Pre-request Script

none form-data x-www-form-urlencoded **raw** binary

```
1 {
2   "id": 2,
3   "nombre": "El laberinto de los espíritus",
4   "autor": "Carlos Ruiz Zafon",
5   "editorial": "Planeta"
6 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text

1 Insertado libro : 2-El laberinto de los espíritus

localhost:8080

GET localhost:8080

Params Authorization Headers (6) Body Pre-request Script

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "nombre": "La sombra del viento",
4   "autor": "Carlos Ruiz Zafon",
5   "editorial": "Planeta"
6 },
7 {
8   "id": 2,
9   "nombre": "El laberinto de los espíritus",
10  "autor": "Carlos Ruiz Zafon",
11  "editorial": "Planeta"
12 }
13 }
14 }
```

4. JWT (JSON WEB TOKEN)

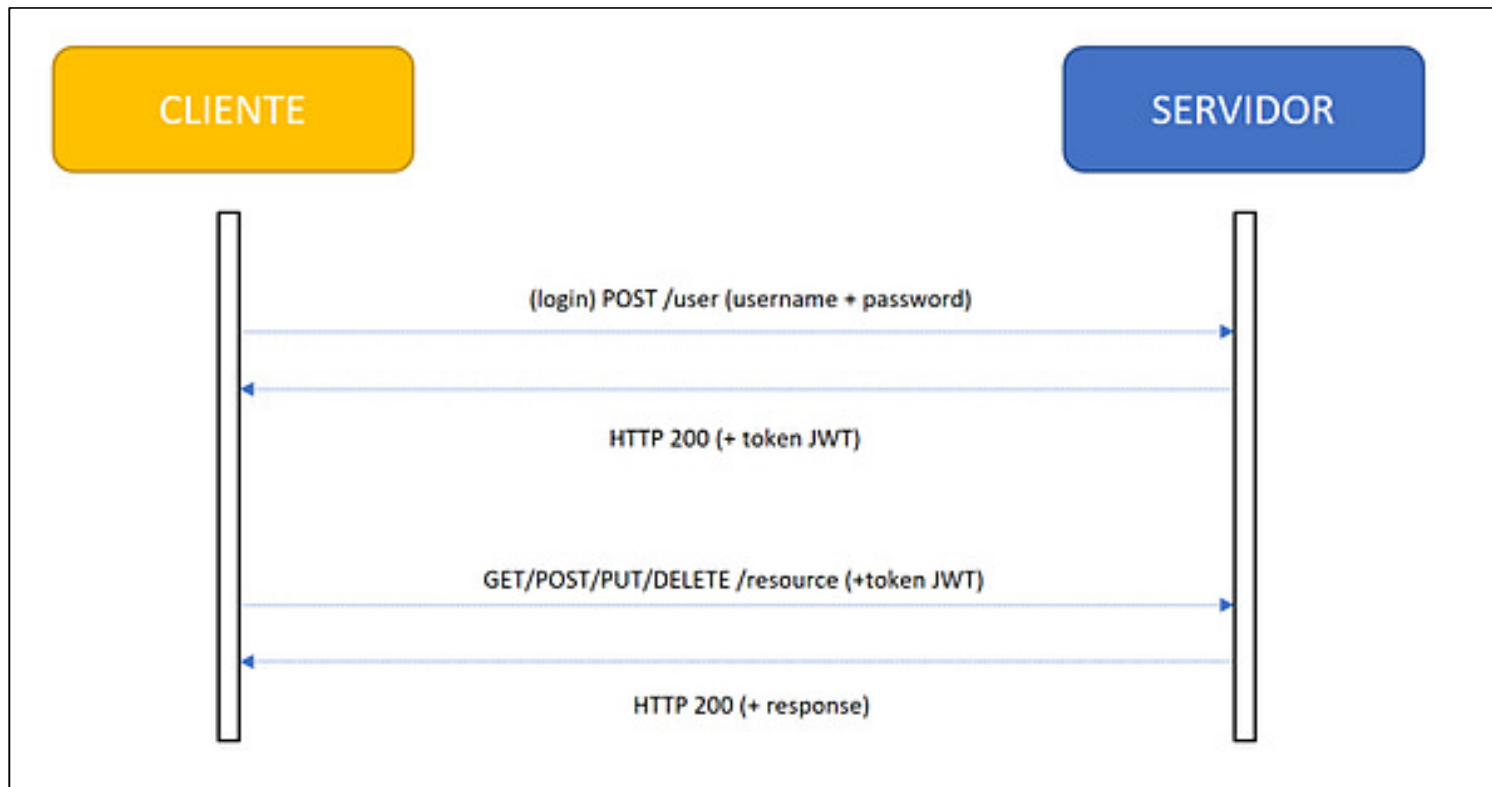
JWT es un estándar de código abierto basado en JSON para crear tokens de acceso que nos permiten securizar las comunicaciones entre cliente y servidor

¿Cómo funciona?

- El cliente se autentica y garantiza su identidad haciendo una petición al servidor de autenticación. Esta petición puede ser mediante usuario contraseña, mediante proveedores externos (Google, Facebook, etc) o mediante otros servicios como LDAP, Active Directory, etc.
- Una vez que el servidor de autenticación garantiza la identidad del cliente, se genera un token de acceso (JWT).
- El cliente usa ese token para acceder a los recursos protegidos que se publican mediante API.
- En cada petición, el servidor descripta el token y comprueba si el cliente tiene permisos para acceder al recurso haciendo una petición al servidor de autorización.

4. JWT (JSON WEB TOKEN)

Son necesarios 3 servidores: el servidor de nuestra API, el servidor de autenticación y el servidor de autorización. No obstante se puede implementar las tres funcionalidades en una única aplicación.



4. JWT (JSON WEB TOKEN)

Estos token están compuestos por tres partes:

Header: contiene el hash que se usa para encriptar el token.

Payload: contiene una serie de atributos (clave, valor) que se encriptan en el token.

Firma: contiene header y payload concatenados y encriptados (Header + "." + Payload + Secret key).

1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyLCJpYXN0IjoiZm9udC5kb290In0.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrZ0ogtVhfEd2o 3

1

Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2

Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

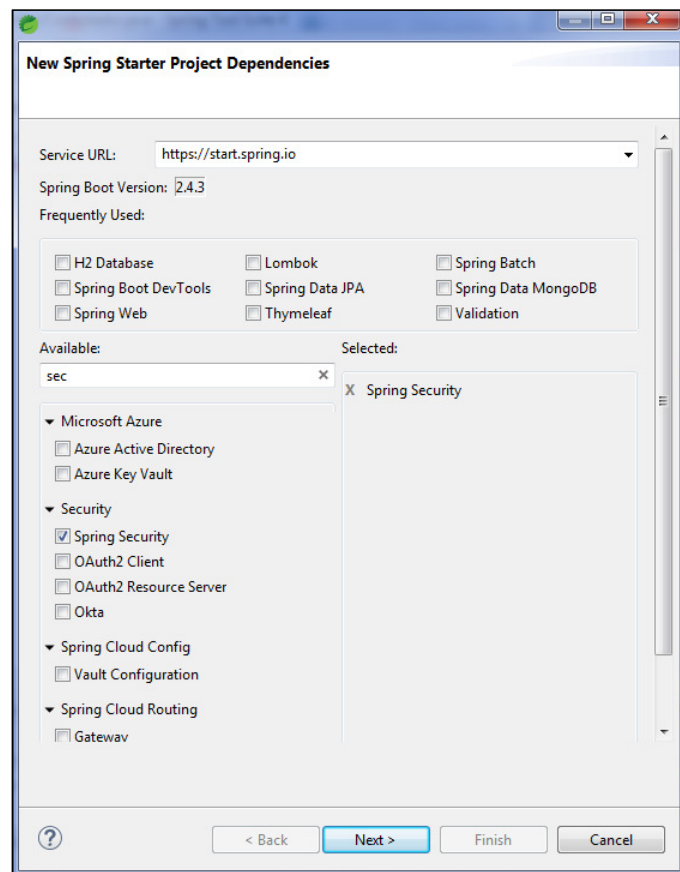
3

Signature

```
HMACSHA256 (
  BASE64URL(header)
  .
  BASE64URL(payload) ,
  secret)
```

4. JWT (JSON WEB TOKEN)

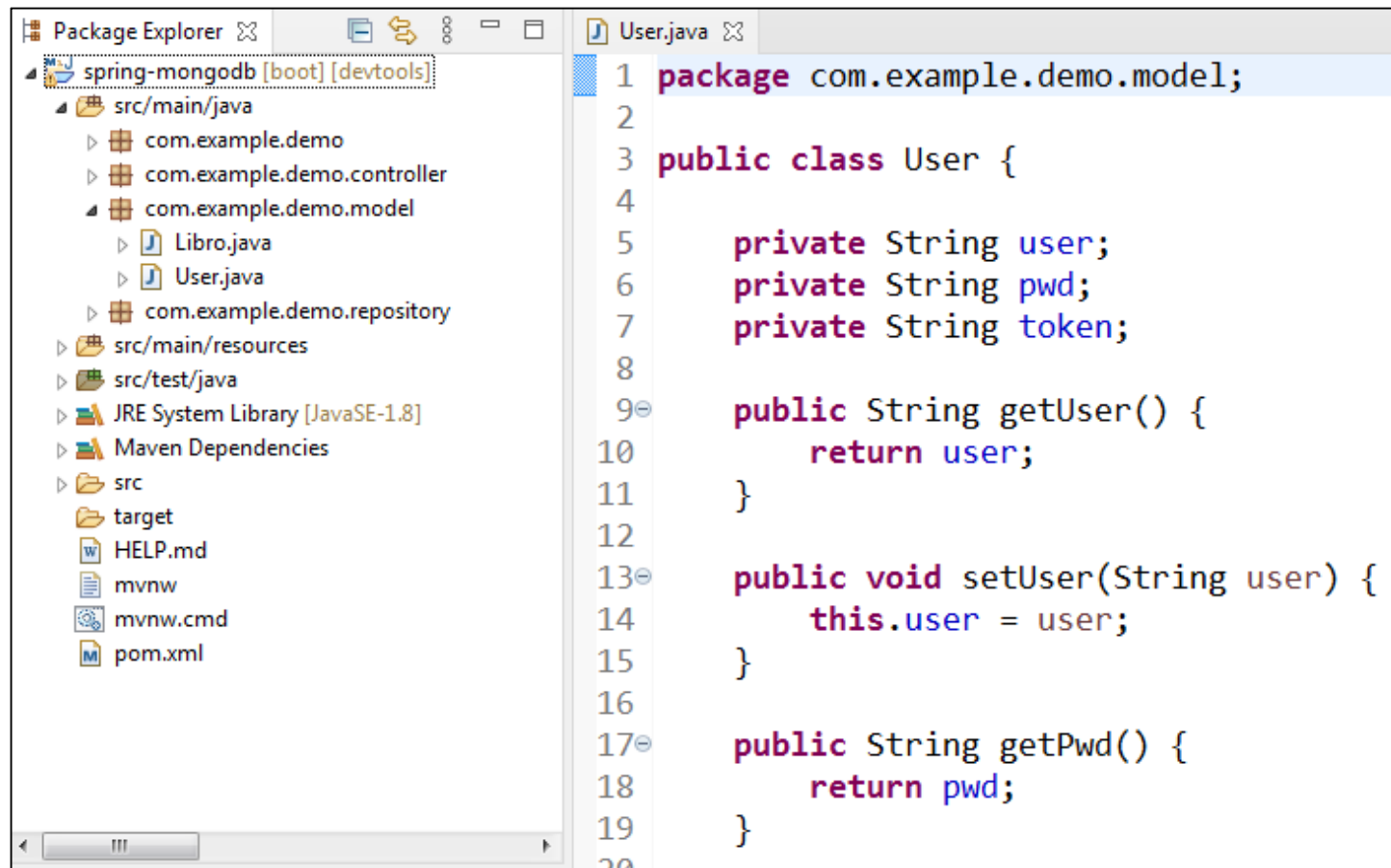
Paso 1) Para agregar seguridad a nuestra aplicación mediante el uso de tokens, primero debemos añadir las dependencias para Spring Security y JWT:



```
</dependency>  
  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
  <groupId>io.jsonwebtoken</groupId>  
  <artifactId>jjwt</artifactId>  
  <version>0.9.0</version>  
</dependency>  
  
</dependencies>
```

4. JWT (JSON WEB TOKEN)

Paso 2) Creamos una clase POJO User, que utilizaremos para el proceso de autenticación:

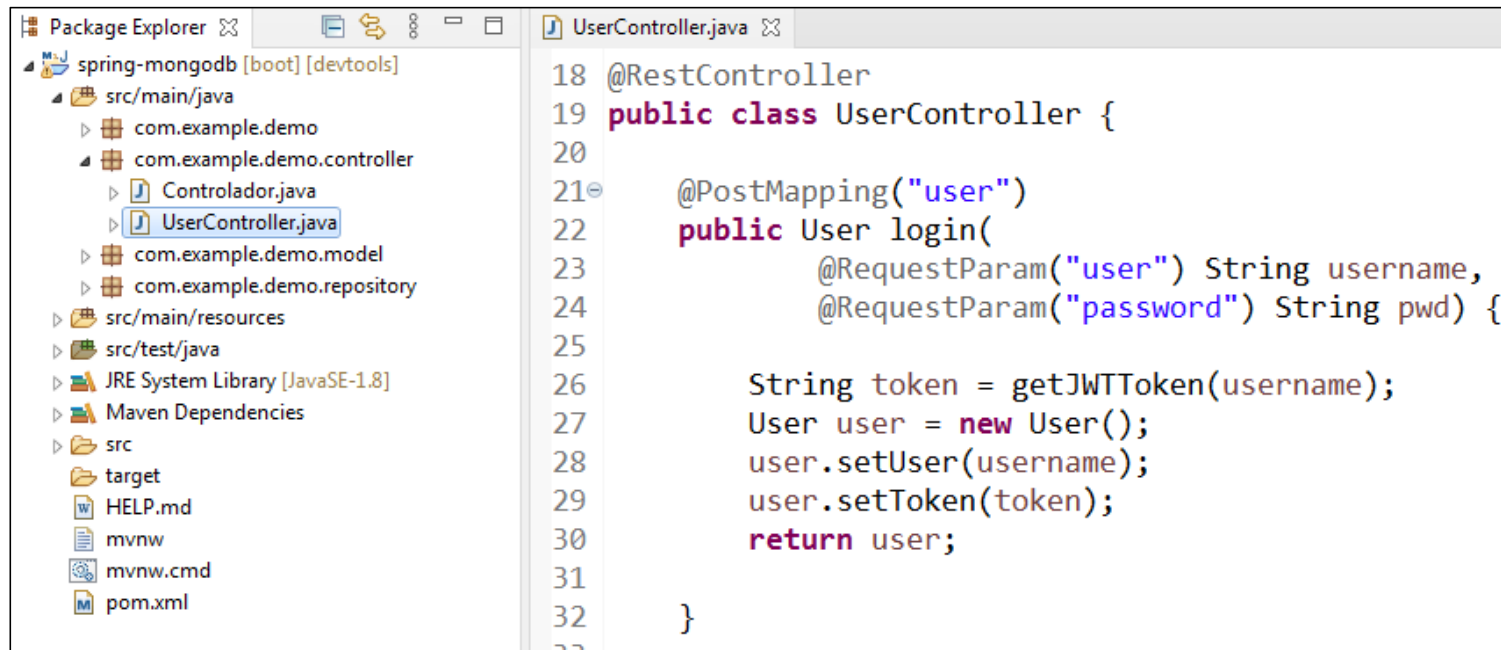


The screenshot shows an IDE with the Package Explorer on the left and the User.java file open in the editor on the right. The Package Explorer shows the project structure for 'spring-mongodb [boot] [devtools]'. The 'src/main/java' directory is expanded, showing the package 'com.example.demo.model'. The 'User.java' file is selected. The editor shows the following code:

```
1 package com.example.demo.model;
2
3 public class User {
4
5     private String user;
6     private String pwd;
7     private String token;
8
9     public String getUser() {
10         return user;
11     }
12
13     public void setUser(String user) {
14         this.user = user;
15     }
16
17     public String getPwd() {
18         return pwd;
19     }
20 }
```

4. JWT (JSON WEB TOKEN)

Paso 3) Vamos a crear otro controlador REST para implementar el proceso de autenticación. El método login intercepta las peticiones POST realizadas a **localhost:8080/user** y retorna un objeto User con el token. En este caso se ofrece un token a todo el mundo, dejando pasar a cualquiera que haga la petición. No realiza ninguna validación de usuario contra una bd (este sería el lugar para ello)



```
18 @RestController
19 public class UserController {
20
21     @PostMapping("user")
22     public User login(
23         @RequestParam("user") String username,
24         @RequestParam("password") String pwd) {
25
26         String token = getJWTToken(username);
27         User user = new User();
28         user.setUser(username);
29         user.setToken(token);
30         return user;
31     }
32 }
```

4. JWT (JSON WEB TOKEN)

Paso 4) El método **getJWTToken** construye el token usando la clase de utilidad *Jwts*, que incluye información sobre su expiración y un objeto **GrantedAuthority** de Spring que usaremos para autorizar las peticiones a los recursos protegidos.

```
private String getJWTToken(String username) {  
    String secretKey = "mySecretKey";  
    List<GrantedAuthority> grantedAuthorities = AuthorityUtils  
        .commaSeparatedStringToAuthorityList("ROLE_USER");  
  
    String token = Jwts  
        .builder()  
        .setId("softtekJWT")  
        .setSubject(username)  
        .claim("authorities",  
            grantedAuthorities.stream()  
                .map(GrantedAuthority::getAuthority)  
                .collect(Collectors.toList()))  
        .setIssuedAt(new Date(System.currentTimeMillis()))  
        .setExpiration(new Date(System.currentTimeMillis() + 600000))  
        .signWith(SignatureAlgorithm.HS512,  
            secretKey.getBytes()).compact();  
  
    return "Bearer " + token;  
}
```

4. JWT (JSON WEB TOKEN)

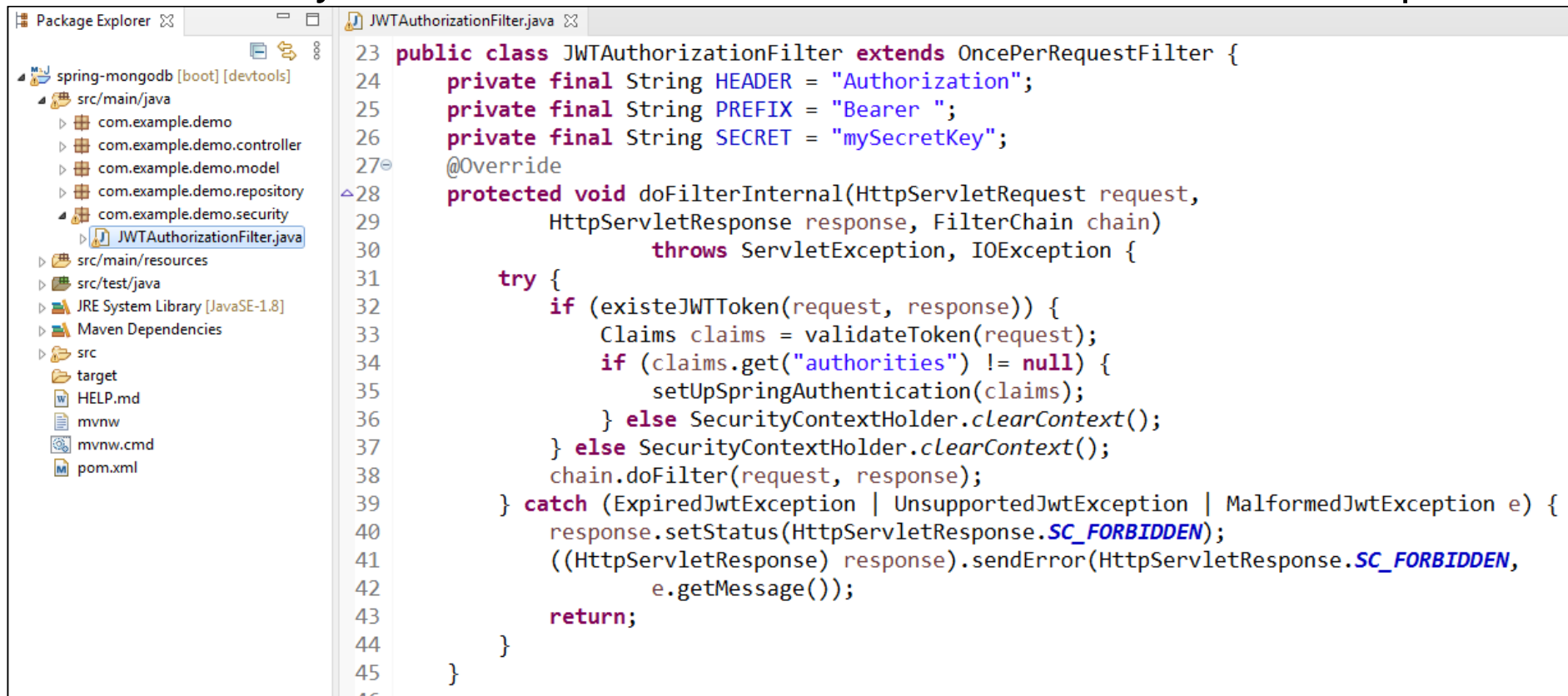
Paso 5) En nuestra clase de arranque añadimos la clase interna **WebSecurityConfig**, que nos permite especificar la configuración de acceso a los recursos publicados.

En este caso se permiten todas las llamadas al controlador `/user`, pero el resto de las llamadas requieren autenticación.

```
SpringMongodbApplication.java
14 @SpringBootApplication
15 public class SpringMongodbApplication {
16
17     public static void main(String[] args) {
18         SpringApplication.run(SpringMongodbApplication.class, args);
19     }
20
21     @EnableWebSecurity
22     @Configuration
23     class WebSecurityConfig extends WebSecurityConfigurerAdapter {
24
25         @Override
26         protected void configure(HttpSecurity http) throws Exception {
27             http.csrf().disable()
28                 .addFilterAfter(new JWTAuthorizationFilter(),
29                               UsernamePasswordAuthenticationFilter.class)
30                 .authorizeRequests()
31                 .antMatchers(HttpMethod.POST, "/user").permitAll()
32                 .anyRequest().authenticated();
33         }
34     }
35 }
```

4. JWT (JSON WEB TOKEN)

Paso 6) Por último, crearemos el filtro **JWTAuthorizationFilter** (extiende de **OncePerRequestFilter**). Permite interceptar todas las invocaciones a los recursos protegidos del servidor, y determinar, en función del token, si el cliente tiene permiso o no.



```
23 public class JWTAuthorizationFilter extends OncePerRequestFilter {
24     private final String HEADER = "Authorization";
25     private final String PREFIX = "Bearer ";
26     private final String SECRET = "mySecretKey";
27     @Override
28     protected void doFilterInternal(HttpServletRequest request,
29                                     HttpServletResponse response, FilterChain chain)
30         throws ServletException, IOException {
31         try {
32             if (existeJWTToken(request, response)) {
33                 Claims claims = validateToken(request);
34                 if (claims.get("authorities") != null) {
35                     setUpSpringAuthentication(claims);
36                 } else SecurityContextHolder.clearContext();
37             } else SecurityContextHolder.clearContext();
38             chain.doFilter(request, response);
39         } catch (ExpiredJwtException | UnsupportedJwtException | MalformedJwtException e) {
40             response.setStatus(HttpServletResponse.SC_FORBIDDEN);
41             ((HttpServletResponse) response).sendError(HttpServletResponse.SC_FORBIDDEN,
42                                                         e.getMessage());
43             return;
44         }
45     }
```

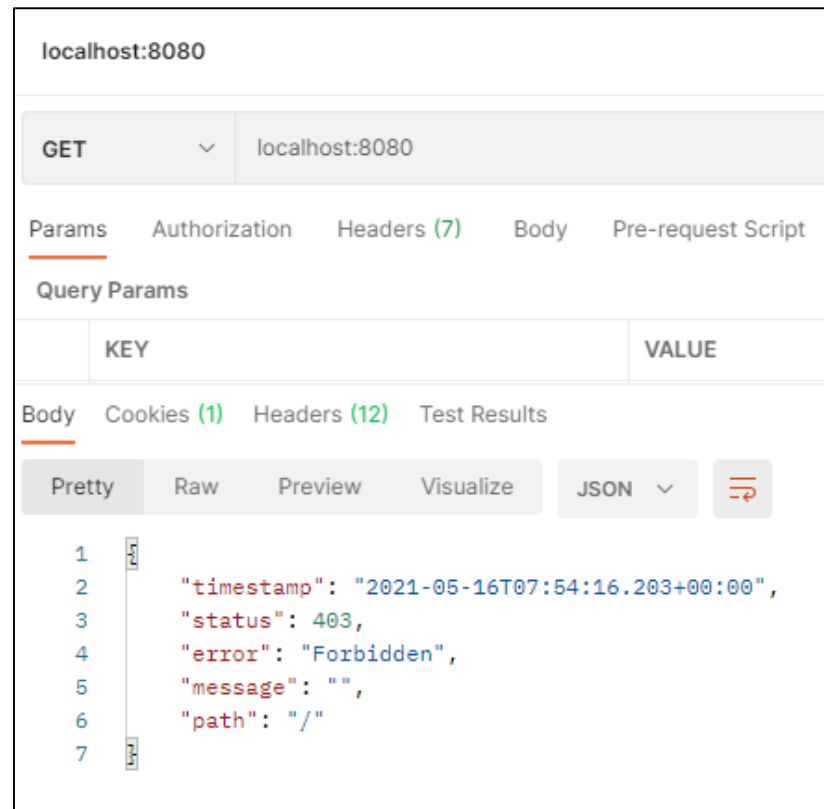

4. JWT (JSON WEB TOKEN)

Paso 7) Este filtro comprueba la existencia del token (**existeJWTToken**). Si existe, lo descripta y valida (**validateToken**). Si está todo OK, añade la configuración necesaria para autorizar la petición (**setUpSpringAuthentication**).

```
JWTAuthorizationFilter.java  ✕
46
47 private Claims validateToken(HttpServletRequest request) {
48     String jwtToken = request.getHeader(HEADER).replace(PREFIX, "");
49     return Jwts.parser().setSigningKey(SECRET.getBytes()).parseClaimsJws(jwtToken).getBody();
50 }
51 //Metodo para la autenticación dentro del flujo de Spring
52 private void setUpSpringAuthentication(Claims claims) {
53     @SuppressWarnings("unchecked")
54     List<String> authorities = (List) claims.get("authorities");
55
56     UsernamePasswordAuthenticationToken auth =
57         new UsernamePasswordAuthenticationToken(claims.getSubject(), null,
58         authorities.stream().map(SimpleGrantedAuthority::new).collect(Collectors.toList()));
59     SecurityContextHolder.getContext().setAuthentication(auth);
60 }
61 private boolean existeJWTToken(HttpServletRequest request, HttpServletResponse res) {
62     String authenticationHeader = request.getHeader(HEADER);
63     if (authenticationHeader == null || !authenticationHeader.startsWith(PREFIX))
64         return false;
65     return true;
66 }
67 }
```

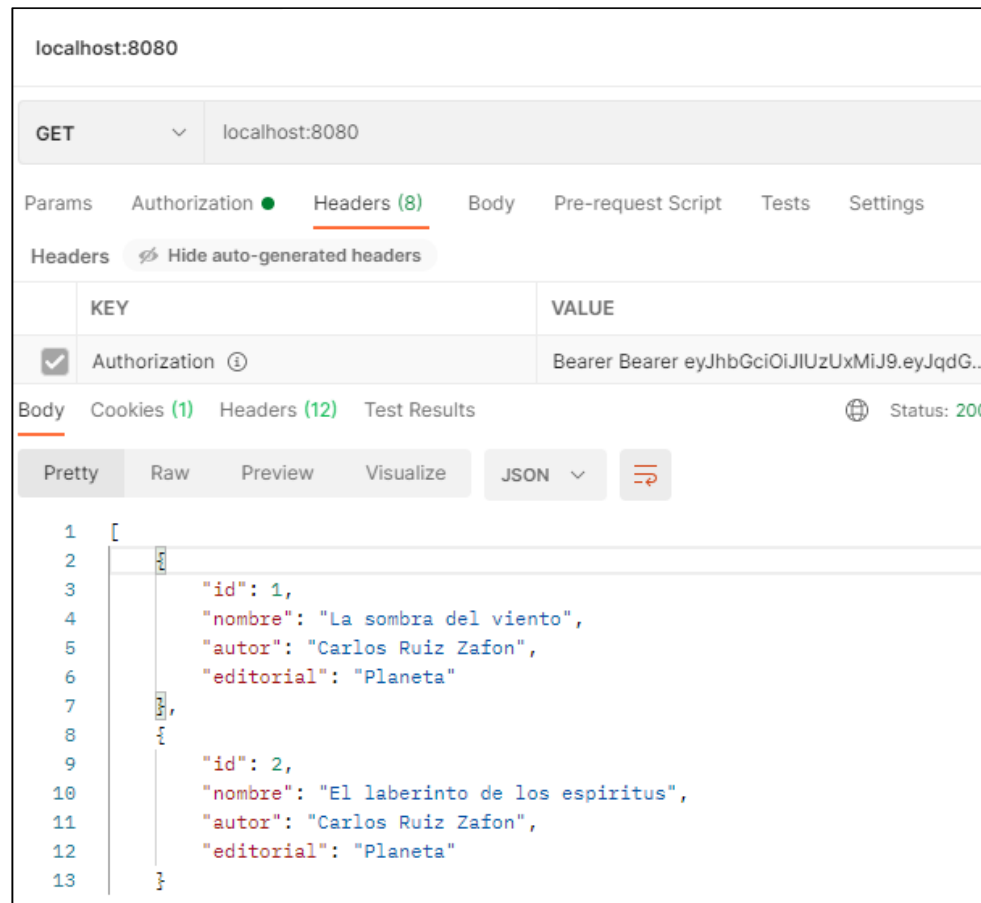

4. JWT (JSON WEB TOKEN)

Paso 8) Reiniciamos la aplicación, y desde Postman hacemos una petición GET a <http://localhost:8080>. Comprobamos que nos devuelve un 403, informando al usuario de que no está autorizado para acceder a ese recurso, que ahora está protegido:



4. JWT (JSON WEB TOKEN)

Paso 10) Con este Token, podemos volver a hacer la petición GET al mismo servicio. Solo debemos incluir una cabecera *Authorization* con el token generado anteriormente.



The screenshot shows a REST client interface for a GET request to localhost:8080. The 'Headers' tab is selected, showing an 'Authorization' header with the value 'Bearer Bearer eyJhbGciOiJIUzUxMiJ9.eyJqdG...'. The 'Body' tab is also visible, showing a JSON response with two book entries.

KEY	VALUE
Authorization ⓘ	Bearer Bearer eyJhbGciOiJIUzUxMiJ9.eyJqdG...

Body: Cookies (1) Headers (12) Test Results Status: 200

```
1 [
2   {
3     "id": 1,
4     "nombre": "La sombra del viento",
5     "autor": "Carlos Ruiz Zafon",
6     "editorial": "Planeta"
7   },
8   {
9     "id": 2,
10    "nombre": "El laberinto de los espíritus",
11    "autor": "Carlos Ruiz Zafon",
12    "editorial": "Planeta"
13  }
14 ]
```



Barcelona
Activa

30
A N Y S
1987 - 2017

barcelona.cat/barcelonactiva