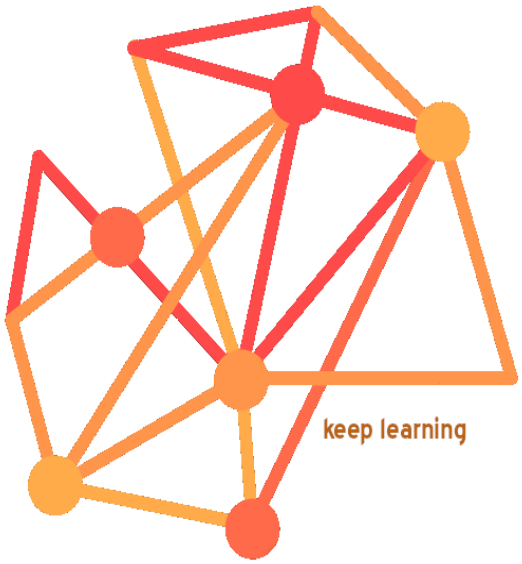
REST con JAVA



Juan Carlos Pérez Rodríguez

Sumario

REST

REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON.

El formato más usado en la actualidad es el formato JSON, ya que es más ligero en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

REST se apoya en HTTP, y mantiene que debe ser un servicio sin estado al igual que HTTP

Los fundamentales principios de REST se basan en separar la API en recursos lógicos. Estos recursos son manipulados usando peticiones HTTP donde el método (GET, POST, PUT, DELETE) tienen un significado específico.

¿Pero qué puede hacer un recurso? Bueno, estos deberían ser sustantivos (no verbos) que tengan sentido desde una perspectiva del consumidor de la API.

Existen tres niveles de calidad a la hora de aplicar REST en el desarrollo de una aplicación web y más concretamente una API que se recogen en un modelo llamado Richardson Maturity Model en honor al tipo que lo estableció, Leonard Richardson padre de la arquitectura orientada a recursos:

1 Uso correcto de URIs

2 Uso correcto de HTTP.

3 Implementar Hypermedia (hateoas)

Además de estas tres reglas, **nunca** se debe guardar **estado** en el **servidor**, toda la información que se requiere para mostrar la información que se solicita debe estar en la consulta por parte del cliente.

Al no guardar estado, REST nos da mucho juego, ya que podemos escalar mejor sin tener que preocuparnos de temas como el almacenamiento de variables de sesión e incluso, podemos jugar con distintas tecnologías para servir determinadas partes o recursos de una misma API.

Nivel 1 REST: Uso correcto de URIs

Cuando desarrollamos una web o una aplicación web, las URLs nos permiten acceder a cada uno de las páginas, secciones o documentos del sitio web.

Cada página, información en una sección, archivo, cuando hablamos de REST, los nombramos como recursos.

El recurso por lo tanto es la información a la que queremos acceder o que queremos modificar o borrar, independientemente de su formato.

Las URL, Uniform Resource Locator , son un tipo de URI, Uniform Resource Identifier, que además de permitir identificar de forma única el recurso, nos permite localizarlo para poder acceder a él o compartir su ubicación.

Una URL se estructura de la siguiente forma:

{protocolo}://{dominio o hostname}[:puerto (opcional)]/{ruta del recurso}?{consulta de filtrado}

Una vez que tienes tus recursos definidos, necesitas identificar qué acciones aplican a ellos y cómo deberían relacionarse con tu API. Los principios REST proveen estrategias para manejar acciones CRUD usando métodos HTTP relacionados de la siguiente forma:

GET /tickets- Devuelve una lista de tickets

GET /tickets/12- Devuelve un ticket específico

POST /tickets- Crea un nuevo ticket

PUT /tickets/12- Actualiza el ticket #12

PATCH /tickets/12- Actualiza parcialmente el ticket #12

DELETE /tickets/12- Elimina el ticket #12

¿El nombre del Endpoint debería ser singular o plural? La regla mantenlo-simple (keep-it-simple) aplica aquí. Si bien tu gramática interna te dirá que está mal describir una instancia única de un recurso utilizando el plural, la respuesta pragmática es mantener el formato de la URL consistente y siempre usar plural. No tener que lidiar con plurales irregulares (person/people) hace más simple la vida del consumidor de la API y es más sencillo para el proveedor de la API implementarla (como los más modernos frameworks manejarán nativamente /tickets y /tickets/12 bajo un controlador común).

¿Pero cómo lidiar con las relaciones? Si una relación puede existir con un sólo recurso, los principios REST proveen una guía útil. Veamos esto con un ejemplo. Supongamos que un ticket consiste en un número de mensajes. Estos mensajes pueden ser lógicamente mapeados al endpoint /tickets de la siguiente forma:

GET /tickets/12/messages - Devuelve una lista de mensajes para el ticket #12

GET /tickets/12/messages/5 - Devuelve el mensaje #5 para el ticket #12

POST /tickets/12/messages - Crea un nuevo mensaje en el ticket #12

PUT /tickets/12/messages/5 – Actualiza el mensaje #5 para el ticket #12

PATCH /tickets/12/messages/5 - Actualiza parcialmente el mensaje #5 para el ticket #12

DELETE /tickets/12/messages/5 - Borra el mensaje #5 para el ticket #12

Filtrado, ordenación y búsqueda en los resultados

Lo mejor es mantener la URL base de recursos tan simple como sea posible. Filtros de resultados complejos, requisitos de ordenamiento y búsqueda avanzada (cuando se limita a un solo tipo de recurso) pueden ser implementados fácilmente como parámetros de consulta en la parte superior de la URL base. Veamos esto en más detalle:

Filtrado:

Usa un único parámetro de consulta por cada campo que implemente el filtro. Por ejemplo, cuando se pide una lista de tickets del endpoint /tickets, podrías querer limitarla a sólo los que están en estado abierto. Esto puede ser logrado con una petición como:

```
GET /tickets?state=open
```

En este caso, state es el parámetro de la consulta que implementa el filtro.

Ordenación:

Similar al filtrado, un parámetro genérico sort puede ser usado para describir las reglas de ordenamiento. Organiza los requerimientos de ordenamiento complejos permitiendo que el parámetro de ordenación sea tomado de una lista de campos separados por coma, cada uno con un posible negativo unario para implicar orden descendiente. Veamos algunos ejemplos:

Devuelve una lista de tickets en orden de prioridad descendiente:

```
GET /tickets?sort=-priority
```

Búsqueda:

A veces los filtros básicos no son suficientes y se necesita la posibilidad de realizar una búsqueda completa sobre el texto. Las consultas de búsqueda deberían ser pasadas directamente al motor de búsqueda y la salida de la API deberían estar en el mismo formato que un lista de resultado normal.

Combinando todo esto, podemos construir consultas como:

GET /tickets?sort=-updated_at – Devuelve los tickets recientemente actualizados

GET /tickets?state=closed&sort=-updated_at – Devuelve los tickets recientemente cerrados

Correcta escritura de los paths de nuestra api

Hemos puesto como ejemplo líneas como esta:

```
GET /tickets/12/messages/5
```

Cuando venimos del lenguaje relacional de las bases de datos la anterior instrucción (de naturaleza jerárquica) nos puede parecer extraña ¿ no sería más lógico buscar el message 5 directamente ?: **GET /messages/5**

En principio podría parecer que es apropiada la línea anterior al usar el lenguaje relacional. Esto es, atacamos directamente a la tabla messages y tomamos con un where la fila con id=5 Sin embargo desde el punto de vista de una aplicación y el interés que pueda tener un usuario no es así. ¿ qué significa para un usuario el message con id 5 ? seguramente poco. Sin embargo sí que tendrá sentido para él que al interponer una solicitud de servicio le hayan asignado un número de ticket: 12 y una vez sabiendo esa información quiera tener una cronología de los mensajes que ha intercambiado con la empresa proveedora a razón de ese servicio registrado con el ticket 12. Así sí tendría sentido preguntar por el quinto mensaje que tuvo con la empresa para el servicio del ticket 12 solicitado.

Adicionalmente a lo anterior Pensemos en la estructura JSON vinculada al ticket 12:

```
{
	ticket: 12,
	messages: [
		{
			message: 1,
			subject: “solicitud de servicio”
		},
		{
			message: 2,
			subject: “Re: le atenderemos el próximo viernes en su domicilio”
		},
		{
			message: 3,
			subject: “Es sábado y no ha venido nadie el viernes como informaron”
		},
		{
			message: 4,
			subject: “Disculpe las molestias resolveremos lo más pronto posible”
		},
	]
}
```

Observar que los objetos de tipo: Message del array anterior en el modelo relacional tendrán a su vez una foreign key hacia el ticket 12. En principio tal información debiera aparecer en el json pero parece redundante. Veamos un ejemplo de un Message así:

```
{
        message: 1,
        subject: “solicitud de servicio”.
        {Ticket: 12,
                messages: [ …]
        }
}
```

Observar el problema que puede llegar a ser si, incluso, a la información del id del ticket mostramos todo el contenido del ticket: Se volvería a mostrar de forma recursiva la información de los mensajes. Para eso hay varias soluciones. Pero la más sencilla es no volver a mostrar el ticket del que depende el mensaje

Al mostrar la información en el JSON de una forma jerárquica (los mensajes son dependientes del ticket) es innecesario volver a mostrar la información del ticket por cada mensaje

Nivel 2: HTTP

Conocer bien HTTP no es opcional para un desarrollador web al que le importe su trabajo.

Para desarrollar APIs REST los aspectos claves que hay que dominar y tener claros son:

- Métodos HTTP

- Códigos de estado

- Aceptación de tipos de contenido

Métodos.

Como hemos visto en el anterior nivel, a la hora de crear URIs no debemos poner verbos que impliquen acción, aunque queramos manipular el recurso.

Para manipular los recursos, HTTP nos dota de los siguientes métodos con los cuales debemos operar:

GET: Para consultar y leer recursos

POST: Para crear recursos

PUT: Para editar recursos

DELETE: Para eliminar recursos.

Por ejemplo para un recurso de facturas.

GET /facturas Nos permite acceder al listado de facturas

POST /facturas Nos permite crear una factura nueva

GET /facturas/123 Nos permite acceder al detalle de una factura

PUT /facturas/123 Nos permite editar la factura, sustituyendo la totalidad de la información anterior por la nueva.

DELETE /facturas/123 Nos permite eliminar la factura

Práctica 1: Detalla como serían las instrucciones que usarías para:
- Obtener el pedido con id 3
- Modificar el pedido con id 7
- Borrar el usuario 2
- Ordenar los pedidos por fecha
- Borrar los pedidos del día: 2020-12-11

Códigos de estado.

Uno de los errores más frecuentes a la hora de construir una API suele ser el reinventar la rueda creando nuestras propias herramientas en lugar de utilizar las que ya han sido creadas, pensadas y testadas. La rueda más reinventada en el desarrollo de APIs son los códigos de error y códigos de estado.

Cuando realizamos una operación, es vitar saber si dicha operación se ha realizado con éxito o en caso contrario, por qué ha fallado.

Un error común sería por ejemplo:

```
Petición
========
PUT /facturas/123


Respuesta
=========
Status Code 200
Content:
{
 success: false,
 code:    734,
 error:   "datos insuficientes"
}
```

En este ejemplo se devuelve un código de estado 200, que significa que la petición se ha realizado correctamente, sin embargo, estamos devolviendo en el cuerpo de la respuesta un error y no el recurso solicitado en la URL.

HTTP tiene un abanico muy amplio que cubre todas las posibles indicaciones que vamos a tener que añadir en nuestras respuestas cuando las operaciones han ido bien o mal.

Es imperativo conocerlos y saber cuándo utilizarlos, independientemente de que desarrolles siguiendo REST.

El siguiente ejemplo sería correcto de la siguiente forma:

```
Petición
========
PUT /facturas/123


Respuesta
=========
Status Code 400
Content:
{
  message: "se debe especificar un id de cliente para la factura"
}
```

Estos son algunos códigos de respuesta HTTP, que a menudo se utilizan con REST:

- *200 OK - Respuesta a un exitoso GET, PUT, PATCH o DELETE. Puede ser usado también para un POST que no resulta en una creación.*
- *201 Created* – [Creada] Respuesta a un POST que resulta en una creación.
- *204 No Content* – [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (como una petición DELETE)
- *304 Not Modified* – [No Modificada] Usado cuando el cacheo de encabezados HTTP está activo
- *400 Bad Request* – [Petición Errónea] La petición está malformada, como por ejemplo, si el contenido no fue bien parseado.
- *401 Unauthorized* – [Desautorizada] Cuando los detalles de autenticación son inválidos o no son otorgados. También útil para disparar un popup de autorización si la API es usada desde un navegador.
- *403 Forbidden* – [Prohibida] Cuando la autenticación es exitosa pero el usuario no tiene permiso al recurso en cuestión.
- *404 Not Found* – [No encontrada] Cuando un recurso no existente es solicitado.
- *405 Method Not Allowed* – [Método no permitido] Cuando un método HTTP que está siendo pedido no está permitido para el usuario autenticado.
- *410 Gone* – [Retirado] Indica que el recurso en ese endpoint ya no está disponible. Útil como una respuesta en blanco para viejas versiones de la API
- *415 Unsupported Media Type* – [Tipo de contenido no soportado] Si el tipo de contenido que solicita la petición es incorrecto
- *422 Unprocessable Entity* – [Entidad improcesable] Utilizada para errores de validación
- *429 Too Many Requests* – [Demasiadas peticiones] Cuando una petición es rechazada debido a la tasa límite .

Cuando todo lo demás falla; En general, se utiliza una **respuesta 500** cuando el procesamiento falla debido a circunstancias imprevistas en el lado del servidor, lo que provoca el error del servidor.

Nivel 3: Hypermedia (hateoas).

Con Hypermedia básicamente añadimos información extra al recurso sobre su conexión a otros recursos relacionados con él.

Aquí tenemos un ejemplo:

```
Request URL: http://miservidor/concesionario/api/v1/clientes/78

Request Method: GET

Status Code: 200 OK
```

Supongamos que obtenemos como respuesta:

```
{
   "id": 78,
   "nombre": "Juan",
   "apellido": "García",
   "coches": [
         {
               "id": 1033
         },
         {
               "id": 3889
         }
   ]
}
```

Con esto ya sabemos que nuestro cliente compró dos coches pero, ¿cómo accedemos a la representación de esos dos recursos?. Sin consultar la documentación del API no tenemos forma de obtener la URL que identifique de forma única a cada uno de los coches. Además, aunque supiésemos conformar la URL de acceso a los recursos, cualquier cliente que quisiese consumir los recursos debería tener la responsabilidad de construir dicha URL. Por último, ¿qué ocurriría si la URL cambiase?, habría que cambiar todos los clientes que consumen los recursos.

Esto se podría solucionar así:

```
{
   "id": 78,
   "nombre": "Juan",
   "apellido": "García",
   "coches": [
         {
               "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/1033"
         },
         {
               "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/3889"
         }
   ]
}
```

De esta forma, ya sabemos dónde debemos ir a buscar los recursos relacionados (coches) con nuestro recurso original (cliente) gracias a la respuesta del servidor (hypertext-driven)

Creación de la aplicación

Preparación del IDE

Vamos a trabajar con Eclipse y con el framework Spring

¿ Por qué ambas elecciones ? Eclipse es un IDE más ligero que Netbeans (aunque menos amable para el no iniciado) Ya estamos en un punto en el que tenemos que profundizar programación de tal forma que usemos más las herramientas que realmente se trabajan en el mercado, más que herramientas de inicio

¿ Por qué Spring ?

Los frameworks nos facilitan trabajo y en el mercado queremos ser lo más productivos posible. La elección de Spring se ha hecho por ser muy popular. Spring se ha hecho especialmente popular por su buena gestión en la inyección de dependencias.

pero...

¿ qué es la inyección de dependencias ?

Veamos que nos dice wikipedia:

Inyección de dependencias (en inglés Dependency Injection, DI) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos. Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar

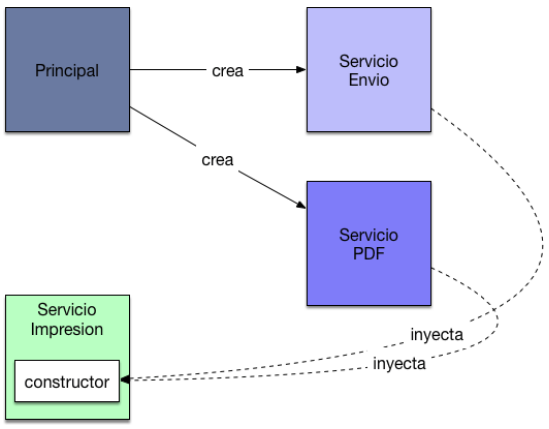
Veamos con un ejemplo:

```
public class ServicioImpresion {
  ServicioEnvio servicioA;
  ServicioPDF servicioB;

  public ServicioImpresion() {

    this.servicioA= new ServicioEnvio();
    this.servicioB= new ServicioPDF();
  }
  public void imprimir() {

    servicioA.enviar();
    servicioB.pdf();
  }
}
```

El código anterior es muy habitual. Como tenemos que hacer uso de las clases ServicioEnvio y ServicioPDF instanciamos objetos de esas clases y los establecemos como atributos de la clase que estamos desarrollando. ¿ Cuál es el problema de esa forma de programar ? Que nosotros debemos conocer sobre la clase que cumpla en la generación de PDF y en servicios de envíos y si surge una nueva también. Tendremos que modificar convenientemente nuestro código cada vez que venga una nueva

¿ No sería mejor que no tuviéramos que tocar esta clase al cambiar la clase que se encarga del servicio de PDF ?

Observar el grafismo:



Si nos fijamos en el grafismo la clase que nosotros estamos desarrollando: ServicioImpresion no hace: new a las otras clases sino que las recibe

```
public class ServicioImpresion {
  ServicioEnvio servicioA;
  ServicioPDF servicioB;

  public ServicioImpresion(ServicioEnvio servicioA,ServicioPDF servicioB) {

    this.servicioA= servicioA;
    this.servicioB= servicioB;
  }
  public void imprimir() {

    servicioA.enviar();
    servicioB.pdf();
  }
}
```

Seguramente un pensamiento que venga es ¿ y qué mejora aporta ? Al fin y al cabo hay que saber de las clases ServicioEnvio y ServicioPDF

Imaginemos que en lugar de conocer el nombre de las Clases lo único que sabemos es de un interfaz que implementan:

```
public interface CrearPDF {
    void pdf();
}


public interface Enviar {
    void enviar();
}
```

Ahora nuestra clase quedaría así:

```
public class ServicioImpresion {
  Enviar servicioA;
  CrearPDF servicioB;

  public ServicioImpresion(Enviar servicioA,CrearPDF servicioB) {

    this.servicioA= servicioA;
    this.servicioB= servicioB;
  }
  public void imprimir() {

    servicioA.enviar();
    servicioB.pdf();
  }
}
```

Nuestra clase ya no tiene idea de que clases son las que le van a hacer el trabajo de enviar o el trabajo de generar pdf Cuando corresponda una parte del programa principal le enviará los objetos que implementen las interfaces pudiendo ser diferentes y nuestra clase seguirá funcionando correctamente

Si disponemos de frameworks como Spring que implementan la inyección de dependencias, nosotros lo único que veremos es una anotación para informar el dato que debemos recibir en nuestra clase:

```
public class ServicioImpresion {
  @Enviarme
  Enviar servicioA;
  @Darmepdf
  CrearPDF servicioB;


  public void imprimir() {

    servicioA.enviar();
    servicioB.pdf();
  }
}
```

Las anotaciones anteriores son inventadas para hacer ver que si tuviéramos un framework que use de la inyección de dependencias y tuviera unas anotaciones de ese tipo: @Enviarme, @Darmepdf ocurriría que haciendo uso del constructor anterior: **ServicioImpresion**(Enviar servicioA,CrearPDF servicioB) nos estaría poniendo en el atributo: servicioA y en: servicioB los objetos que precisa nuestra clase Observar que de esta forma si el framework en el futuro quisiera usar otras clases diferentes para crear un pdf a nosotros no nos afectaría para nada e independizaríamos más nuestro código Si lo pensamos un poco no difiere de como son las clases con anotaciones JPA Nosotros no vemos "el trabajo por detrás" simplemente anotamos las clases, atributos y métodos y JPA ya se encarga del trabajo con la base de datos

Plugins para Eclipse

Nota: Necesitamos un **jdk 17** o superior para la versión de spring que vamos a usar

Al final de la instalación de Eclipse entramos (creamos el workspace) y una vez dentro nos vamos al market:

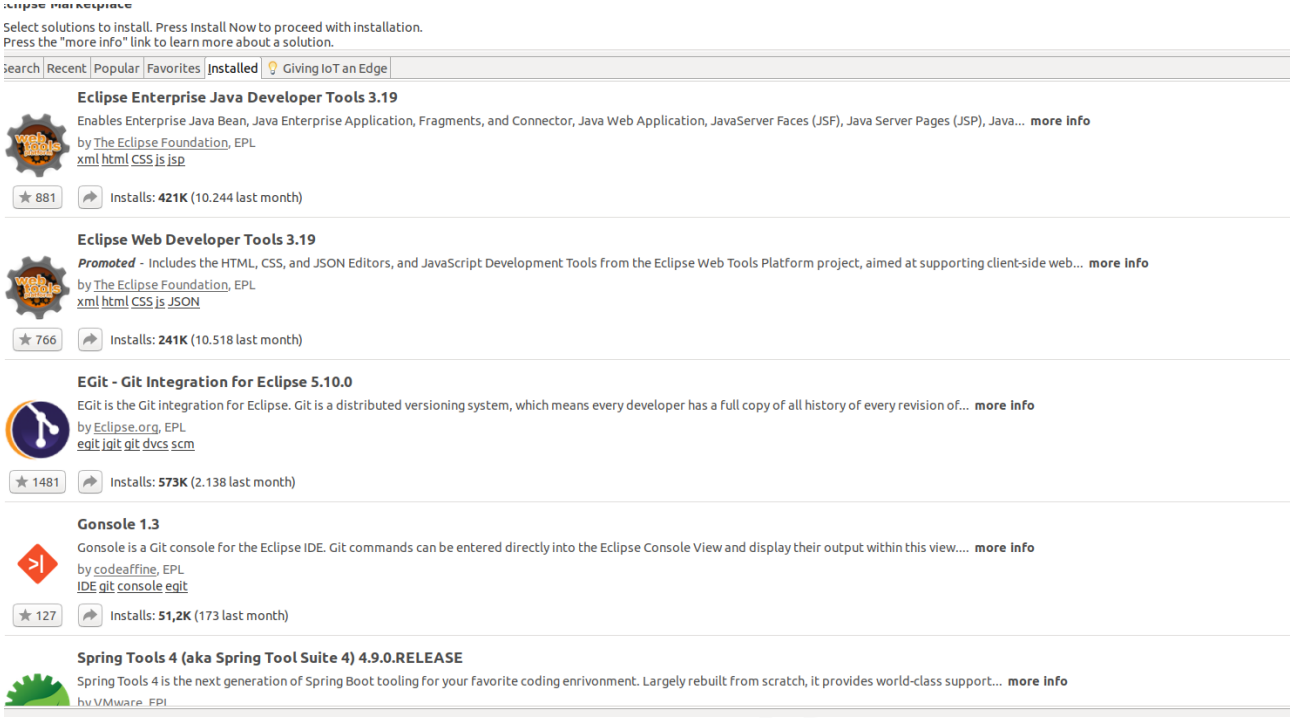
help-→eclipse marketplace

Instalamos:

- spring tools

- Eclipse web developer tools

Ejemplo de una posible configuración de instalación del marketplace:



Ahora para crear un proyecto Spring nos vamos a apoyar en

Spring Initializr: https://start.spring.io/

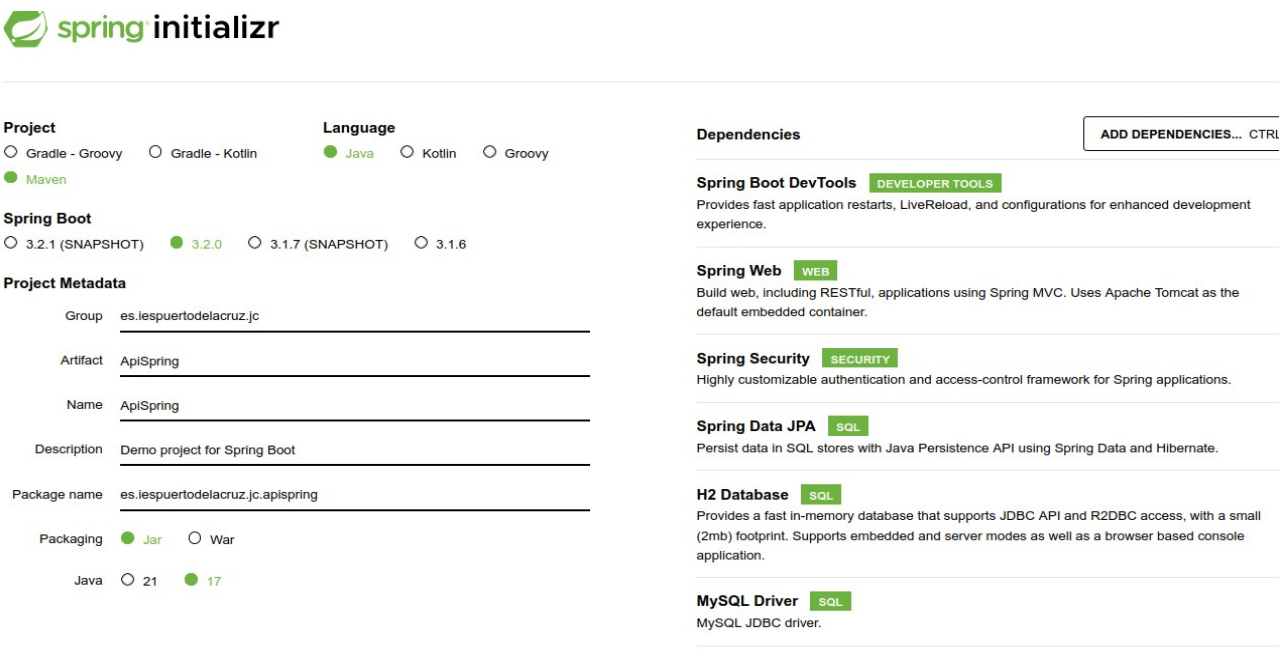
Que nos permite hacer un proyecto spring boot con lo que necesitemos:

ponemos el en grupo la base: es.iespuertodelacruz.nombrealumno

El nombre en Artifact es el del proyecto

elegimos jar, java 17, maven project

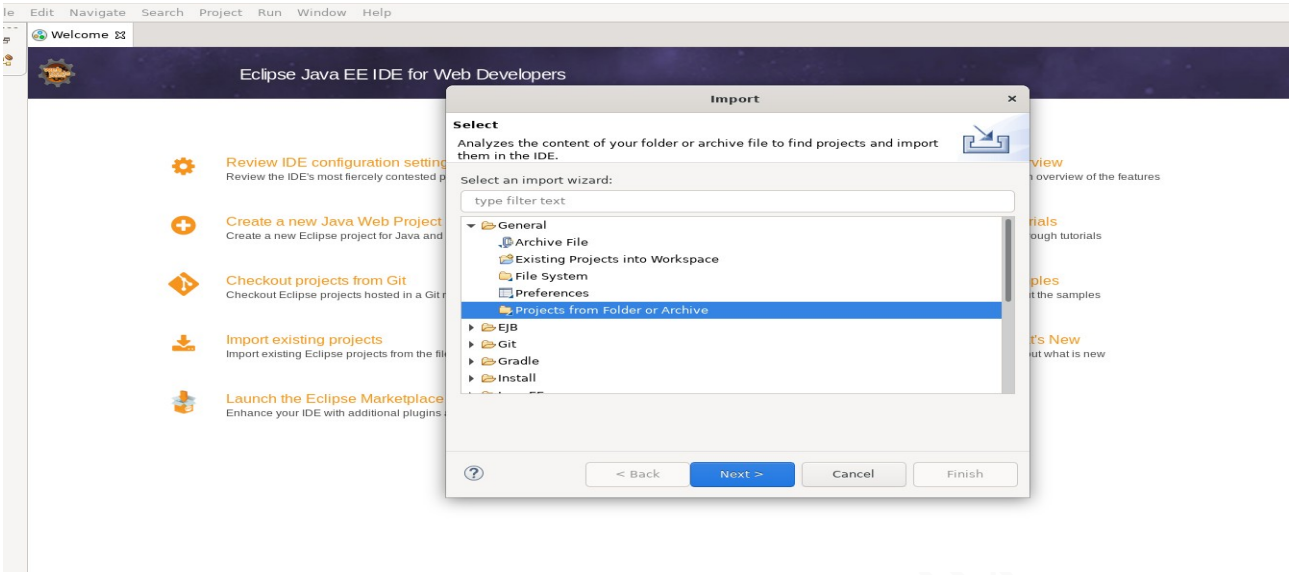
en dependencias: Web, Data JPA, Mysql driver, Security, DevTools, H2 Database



Mysql driver ya que usaremos como persistencia. H2 es otro driver para persistencia (al cargar en memoria es buena para test) JPA para usar un ORM, Spring web nos habilita REST y cualquier aplicación web, Security nos habilitá limitar acceso si autenticado y devtools para que nos autocargue la aplicación cada vez que guardemos

Una vez descargado y descomprimido desde Eclipse:

File→ import → General → project from folder or archive



Nos falta incluir Swagger (realmente openapi), que no nos viene en Spring. Y el manejo del token Jwt (para la parte de seguridad que veremos más adelante) También agregamos Jackson que da soporte para XML, JSON,… Lo pondremos agregando en el fichero pom.xml las dependencias:

```
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springdoc</groupId>
      <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
      <version>2.0.2</version>
   </dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->
<dependency>
       <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

El código es funcional, pero si hubiera problemas (solicita url para el entitymanagerfactory etc) podemos pasarle algunas propiedades de conexión a base de datos (luego no las necesitaremos)

src/main/resources/application.properties :

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/seguimientomonedas?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=1q2w3e4r
logging.level.org.hibernate.SQL  =debug
logging.level.es.iespuertodelacruz.jc.monedas=DEBUG
spring.jpa.show-sql=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.mvc.pathmatch.matching-strategy = ant-path-matcher
```

Si queremos que esté activa la consola de H2 (es una especie de phpmyadmin para H2) agregaremos a lo anterior:

```
spring.h2.console.enabled=true
spring.h2.console.path=/console
```

Ahora botón derecho sobre el proyecto → Run as → Spring boot app

Al ejecutar debemos NO tener errores. Pero queremos tener una salida apropiada. Ahora mismo si accedemos a: localhost:8080 veremos que todo nos lleva a un login

Primero ponemos una configuración que no nos active la seguridad (ya la habilitaremos más tarde)

Creamos un package: **config** y dentro una clase con un nombre significativo de que vamos a quitar la seguridad: ApplicationNoSecurity.java

Ponemos dentro el código:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCust
omizer;

@Configuration
public class ApplicationNoSecurity {

    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring()
            .requestMatchers("/**");
    }

}
```

Mediante: @Configuration le decimos a Spring que es una clase para configuración. Luego los @Bean son los objetos que podemos inyectar (luego veremos otros que también se pueden inyectar) En definitiva lo que estamos haciendo es darle un objeto personalizado de los que usa Spring cuando tiene activada la seguridad: un WebSecurityCustomizer Este objeto se arranca al inicio (ficheros de configuración) y dentro vemos lo realmente importante:

```
return (web) -> web.ignoring()
            .requestMatchers("/**");
```

Que está diciendo que se ignore: web.ignoring() cualquier solicitud que coincida con el patrón: reqquestMatchers(“/**”) Evidentemente ese patrón cuadra con cualquier solicitud. Así que de facto se está desactivando la seguridad

¡cuidado! Con algunas versiones de spring security hay problemas con la consola de h2. Sabemos que antes la hemos desactivado en las properties de spring. Sin embargo si tuviéramos algún problema podemos cambiar el scope de H2 y que únicamente se ejecute con los test (siempre que la estemos usando únicamente para test) Lo que quedaría entonces en el: pom.xml sería:

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

                  <scope>test</scope>

                  <!--
                  <scope>runtime</scope>
                -->
            </dependency>
```

Cuando usamos la anotación @Configuration estamos estableciendo el contexto de componentes (objetos Bean) que se van a inyectar.

Ahora ya no debiera aparecer la ventana de login. Pero queremos ver que cuando hagamos una petición nos devuelva algo.

Para las versiones que hay problemas con H2 tenemos otra forma de solucionar (sin tener que desactivar la consola de h2) Le diremos en ese caso específicamente las rutas separadas que no se deben ver afectadas por la seguridad:

```
@Configuration
public class ApplicationNoSecurity {

    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring()
            .requestMatchers(new AntPathRequestMatcher("/h2-console/**"))
            .requestMatchers(new AntPathRequestMatcher("/api/**"))
            //.requestMatchers("/**");
            ;
    }

}
```

En el ejemplo queda desactivada la seguridad en todas las subrutas de /api y las de /h2-console

Observar que para que esté operativa la consola de h2 implica dejar el application.properties sin las dos líneas que habíamos visto: `spring.h2.console.enable  spring.h2.console.path`

Práctica 2: Reproducir el código que deje funcional y “desactive” temporalmente la seguridad. Comprobar que permite acceder a la consola de h2__¿ si accedes a la ruta raíz muestra alguna página. Y si es así por qué ?

Nuestro primer controlador

Vamos a crear un fichero: PruebaController.java (en la propia carpeta donde está el Application.java) y ponemos:

```
@RestController
@CrossOrigin
@RequestMapping("/api/personas")
public class PruebaController {

    @GetMapping //si queremos subruta lleva paréntesis
    public ResponseEntity<?> getAll(){

        return ResponseEntity.ok("Esta respuesta es de prueba ");
    }

}
```

El código anterior nos dice que es un controller para una api REST: @RestController

Para evitar los bloqueos de protección de CSRF (por ejemplo cuando se llama a la api desde una app react en la misma máquina) ponemos: @CrossOrigin que permite cualquier solicitud de otro origen web y Finalmente la ruta base a la que se va a responder desde esta clase:

@RequestMapping("/api/personas") Con esta anotación le estamos diciendo que todas las rutas que responda esta clase serán a un path que incluya: /api/personas. Por ejemplo: /api/personas/23

Por otro lado al usar RequestMapping en lugar de GetMapping o PostMapping le estamos diciendo que aceptar tanto peticiones GET como POST

Lo siguiente que observamos es el método getAll():

```
    @GetMapping //si queremos subruta lleva paréntesis
    public ResponseEntity<?> getAll(){
        return ResponseEntity.ok("Esta respuesta es de prueba ");
    }
```

@GetMapping está indicando que el método getAll() responde a las peticiones GET y como no se establece ningún subpath, indica que responde a todas las GET que se hagan a /api/personas

Si se hubiera querido que fuera a un subpath: Por ejemplo: /api/personas/adultas entonces tendríamos que haber puesto: @GetMapping("/adultas")

Observar que únicamente se pone la parte del subpath, ya que a nivel de la clase se ha establecido que todas las rutas incluirán: /api/personas ahora al poner en el método "/adultas" se le agrega a la anterior ruta: /api/personas/adultas

Práctica 3: Reproducir el código anterior. Ejecutar desde postman o rested la petición localhost:8080/api/personas y comprobar que devuelve el mensaje

Nos falta entender el return: `return ResponseEntity.ok("Esta respuesta es de prueba ");`

Con spring tenemos una clase muy interesante para las respuestas HTTP, que es ResponseEntity Cuando queramos códigos de éxito (200) usamos ResponseEntity.ok y dentro le pasamos el objeto (puede ser cualquier objeto de nuestro modelo) que queramos que muestre al usuario en el cuerpo del mensaje HTTP que le devolvemos. Típicamente lo va a mostrar como un JSON

Configurar Swagger/Openapi

Documentar una api es fundamental. No tiene sentido una api si no hay clientes que la puedan usar. Así que es muy útil facilitar el uso de la api a los posibles clientes. Swagger ha sido una herramienta que ha permitido documentar una api y facilitar que los clientes puedan usarla. En versiones 3.0 y posteriores de spring se utiliza OpenApi para la misma labor. Generaremos en el paquete config un fichero (por ejemplo lo podemos llamar: SwaggeConfig.java) para configurar openapi:

```
@Configuration
@OpenAPIDefinition(info = @Info(title = "Api Gente", version = "v1"))
public class SwaggerConfig{

      @Bean
      public OpenAPI customizeOpenAPI() {
          final String securitySchemeName = "bearerAuth";
          return new OpenAPI()
            .addSecurityItem(new SecurityRequirement()
              .addList(securitySchemeName))
            .components(new Components()
              .addSecuritySchemes(securitySchemeName, new
io.swagger.v3.oas.models.security.SecurityScheme()
                .name(securitySchemeName)
                .type(io.swagger.v3.oas.models.security.SecurityScheme.Type.HTTP
)
                .scheme("bearer")
                .bearerFormat("JWT")));
    }
}
```

El fichero de configuración establece que se usará autenticación por token (típicamente los token jwt en este caso van precedidos de la palabra: “bearer” en la cabecera)

Ahora para acceder a la documentación iremos a la url:

```
http://localhost:8080/swagger-ui/index.html
```

Práctica 4: Reproducir el código de openapi. Acceder a la url dada para openapi/swagger ¿ documenta ? ¿ qué urls quedan expuestas ? Toma captura de pantalla

Capas de una api rest tradicional

Bien, ya tenemos los inicios de una api rest.

Es habitual que se quiera acceder a la infraestructura (una base de datos) para obtener la información o almacenarla según las peticiones que se hagan a la capa controlador. Vamos a ver un modelo que usa las siguientes capas:

- entity

- repositorio

- servicio

- dto

- controlador

La capa controladora ya la hemos visto (el fichero PruebaController.java) su objetivo es interactuar con el exterior mediante urls Las otras capas están relacionadas con los objetos de nuestra lógica de negocio y como se almacenan o sirven

Vamos a ir introduciendo las capas de más próximas a la DDBB hasta llegar al Controller

Para ello precisaremos dar información de base de datos en application.properties. Se muestra un ejemplo (se deberá cambiar al nombre de la DDBB: seguimientomonedas, el usuario y password: root, 1q2w3e4 y más adelante los logging.level para hacer debug)

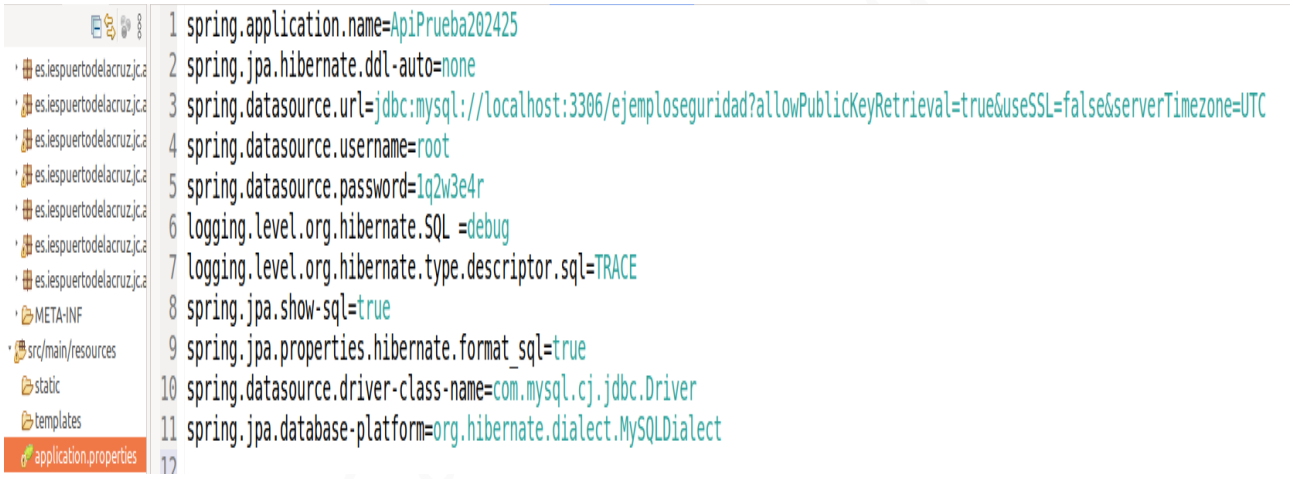
src/main/resources/application.properties :

```
spring.h2.console.enabled=false # linea opcional a true/false


spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/seguimientomonedas?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=1q2w3e4r
logging.level.org.hibernate.SQL=debug
logging.level.org.hibernate.type.descriptor.sql=TRACE
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
logging.level.es.iespuertodelacruz.jc.monedas=DEBUG
logging.level.es.iespuertodelacruz.jc.monedas.controller=DEBUG
```

Nota:

url=jdbc:mysql… nos da la ruta a la base de datos: ip, puerto, nombreDDBB, y los parámetros a partir del interrogante: useSSL=false (no activamos ssl al ser en localhost) y serverTimezone para especificar la zona horaria. Los logging.level son para poder ejecutar un logger y ver info en consola



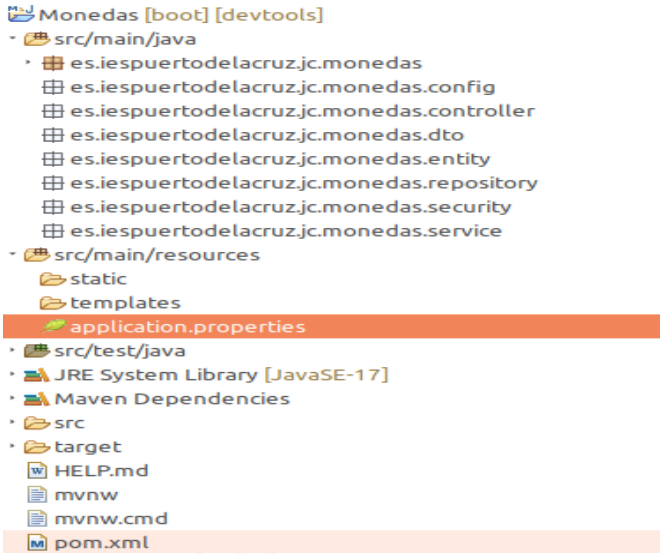
Práctica 5: Realizar lo explicado para crear un proyecto nuevo spring para api rest. Ejecutarlo mediante: botón derecho sobre el proyecto → run as → spring boot app Comprobar que compila sin error y queda lanzado. Modificar el main() para que informe en consola (logger.info()) de que la aplicación ha sido lanzada así como que aparezca tu nombre. Comprobar que al modificar y guardar automáticamente vuelve a lanzar la aplicación (eso es por las devtools que pusimos como dependencia)
Nota: el logger funcionará si en application.properties hemos puesto el nombre del paquete como accesible a debug

Separación en paquetes

Crearemos como mínimo los paquetes:

entity, repository, service, controller, security, dto, config

Es recomendable el paquete DTO para separar las clases que mapean las tablas reales de los objetos que se muestran a los usuarios del servicio REST



Creación de Entities de forma automática

SEGUIREMOS EL PROCEDIMIENTO PARA CONVERTIR A JPA UN PROYECTO ECLIPSE Y OBTENER ENTITIES FROM TABLES (anexo o tomado de otros dossier)

Nota: si trabajamos con jakarta el wizard puede fallar ya que hace un import de todo el paquete: javax.persistence.* En ese caso, simplemente hacer los import pertinentes de jakarta en lugar de javax

Nota2: El fichero persistence.xml generado se puede establecer y hacer las optimizaciones, pero puede eliminarse y seguirá funcionando la app. La queja por las entities no listadas se resuelven con exclude-ulisted-classes:

```
<exclude-unlisted-classes>false</exclude-unlisted-classes>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" x
    <persistence-unit name="Almacentienda">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
```

De forma anterior ya **obtendremos el contenido del package: entity**

El patrón repositorio

Es típico hacer un CRUD(Crear, Leer, Actualizar, Eliminar) con almacén de datos. Mediante las entities tenemos un mapeo objeto-relacional que nos permite trabajar con clases sin acudir a jdbc y SQL . Llamaremos capa de mapeo de datos a nuestras entities (Data Mapper)

El patrón repositorio media entre el resto de nuestro proyecto y las capas de mapeo de datos, actuando como una colección de objetos de dominio en memoria ¿ y para qué esa capa adicional ?

Una vez hemos montado nuestra capa de “Data Mapper”, al ir construyendo el resto de la aplicación, nos vamos dando cuenta de que necesitamos crear un método que acepte condiciones y filtros para realizar un gran número de consultas diferentes. Por ejemplo, necesitamos listados de usuarios paginados, filtrados de usuario por edad, etc

Pongámonos en nuestro caso en concreto. Tenemos Entities mapeadas a una base de datos. Es muy probable que querramos un listado de todos los objetos de una tabla. Imaginemos la tabla productos Habrá que construir un método findAll() para que nos devuelva un listado de la clase producto. Lo mismo pasará con user, etc.

Spring se apoya en el patrón repositorio y es conocedor de los métodos habituales que se quieren de nuestros objetos de la base de datos. Por ejemplo, conoce de métodos de paginado de objetos, borrado, etc

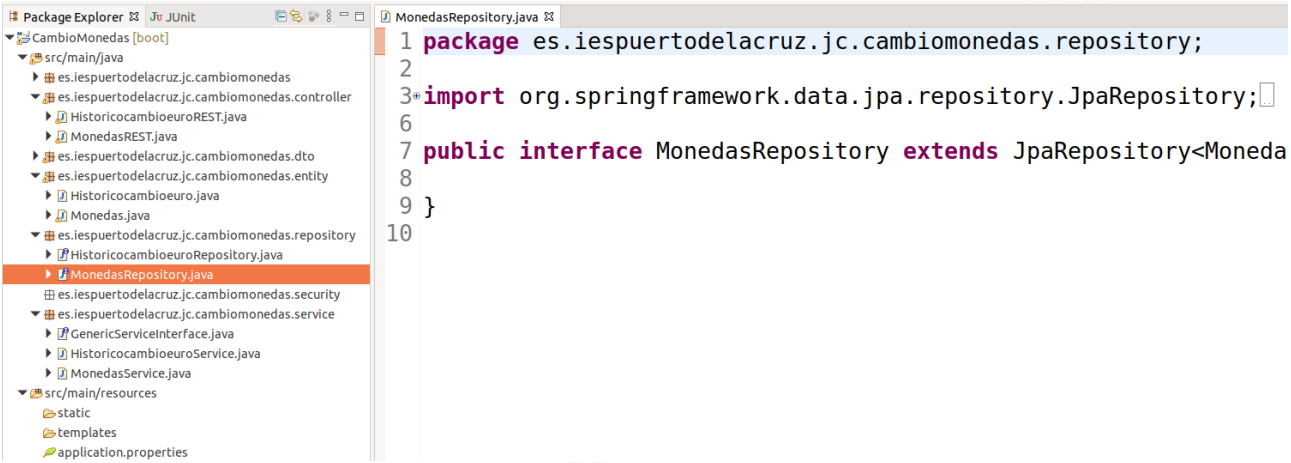
Para aprovechar esas ventajas de Spring únicamente tenemos que crear un interfaz que implemente JpaRepository<T,E> Donde T es la entity y E es el tipo de identificador (primary key) que usemos. Veamos un ejemplo. Para una Entity llamada Producto con una clave principal de tipo Integer haríamos:

```
@Repository
public interface ProductoRepository extends JpaRepository<Producto, Integer> {

}
```

La anotación de @Repository no es del todo necesaria porque al extender de JpaRepository ya se tiene esa información. Ahora bien, aprovechamos para entender que al poner @Repository estamos en un caso parecido a como poníamos @Bean: son objetos que vamos a poder inyectar y

pero son objetos específicos (permiten acceso a nuestra infraestructura de DDBB) y los anotaremos como tales para luego obtenerlos allí donde los necesitemos mediante la anotación @autowired

Veamos ejemplo:



Con lo anterior Spring ya se encarga de crearnos métodos tan útiles como: findAll(), nosotros no tenemos que hacer nada: **Obteniendo así el package: repository**

Spring hace uso también de otro patrón (patrón fachada) que lo utiliza en el concepto de: servicio. También tenemos que entenderlo

Veamos la definición del patrón fachada según wikipedia:

Servicios y Patrón fachada

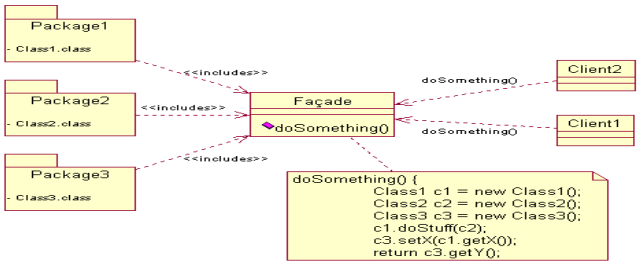
Podemos entender muchas veces los servicios como nuestros **casos de uso**. Así por ejemplo, en un caso de un aplicativo de empresa, la relación de asesoramiento entre empleados y clientes (muchos a muchos en base de datos) implica dos entity: Empleado, Cliente y el servicio podría ser: AsesoramientoService este servicio incluiría hacer uso de ambas entities y tendría métodos del tipo: consejosFiscales(), consejosInversion(), etc

Ahora pongámonos en otro ejemplo. En nuestro aplicativo hay que obtener documentos en pdf (informes) y también otros en formato xml etc. Bueno, eso implicará probablemente utilizar librerías de terceros para generar pdf. Una librería de terceros está pensada para casos generales siendo así muy extensa, con múltiples opciones y tendremos que invertir tiempo en aprender como funciona ese servicio. Para nuestra empresa sin embargo únicamente necesitaremos unas pocas funciones y probablemente necesitarán una adaptación/particularización de nuestro caso. Una vez hayamos desarrollado el aplicativo pasan años y otro desarrollador de la empresa vuelve a necesitar hacer uso de los servicios de pdf. Otra vez tendrá que volver a aprender como funciona para adaptarlo a los requisitos de nuestra empresa

¿ y qué tal si desde el primer momento se construyera una clase específica (fachada) que envolviera la librería de terceros de pdf para que únicamente aparezcan las 4, 5 funciones que realmente usamos en nuestra empresa ? Vamos a ponerle el nombre: NuestroPdfService De hacerlo, la segunda vez que haya que hacer uso de la librería de pdf, únicamente tengamos que mirar las pocas funciones que aparezcan en NuestroPdfService, ya que están adaptadas a las necesidades de nuestra empresa. Se ahorrará tiempo. Este es un caso del patrón fachada

Fachada (Facade) es un tipo de patrón de diseño estructural. Viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos.

En general podemos asociar los Ser



Si nos fijamos en el grafismo anterior, vemos que nos permite “aislar” de las clases 1,2,3 a nuestros clientes. Puede haber modificaciones en esas clases que no afectará a Client1, Client2. Ese es uno de los motivos del patrón fachada. Otro motivo puede ser para proyectos grandes poder separar más nuestro código aumentando el desacoplamiento

Spring está pensado para trabajar con proyectos grandes. Así que los service y el patrón fachada tiene completo sentido.

En nuestro caso, y en general en proyectos pequeños prácticamente veremos que reproducimos el repositorio sin más variación a un servicio.

Observar este código:

```
public interface IGenericService<T,E> {
	Iterable<T> findAll();
	Optional<T> findById(E id);
	T save(T element);
	void deleteById(E id);

}
```

Imaginemos que lo hemos puesto en nuestro paquete service

Ahora si queremos crear un Servicio (ejemplo para la app de la tienda con tabla productos):

```
@Service
public class MonedasService implements IGenericService<Moneda, Integer>{

	@Autowired
	private IMonedaRepository monedaRepository;

	@Override
	@Transactional(readOnly=true)
	public Iterable<Moneda> findAll() {
		return monedaRepository.findAll();
	}

	@Override
	@Transactional(readOnly=true)
	public Optional<Moneda> findById(Integer id) {
		return monedaRepository.findById(id);
	}
```

```
	@Override
	@Transactional
	public Moneda save(Moneda element) {
		return monedaRepository.save(element);
	}

	@Override
	@Transactional
	public void deleteById(Integer id) {
		monedaRepository.deleteById(id);

	}

}
```

Vemos que no se aporta mucho respecto a lo que hace un repositorio. Lo que sí que es importante que señalemos son las notaciones Spring nuevas:

@Service Nos permite que Spring considere un servicio la clase (NOTA: Spring lo trata como componente (@component) que podemos inyectar pero adicionalmente le da trato de servicio)

@Autowired Hace que cargue automáticamente un objeto (lo inyecta) en este caso el repositorio (al igual que con el @service, @repository te toma un componente como un inyectable repositorio)

@Transactional Cuando trabajamos con Entities tenemos que hacer las gestiones en un ambiente transaccional. Cuando explicitamos readonly no se considera que vayamos a modificar en la base de datos. En otro caso se considera que sí será una transacción que admite cambios en la base de datos. No olvidar hacer uso de @Transactional !!!

Habremos de hacer los servicios para cada una de las entities. Veamos los dos servicios para monedas

Práctica 6: Crear los Repositorios y servicios de nuestra api (no olvidar las anotaciones @Repository y @Service porque no funcionaría)

Patrón DTO (Data Transfer Object)

El patrón DTO tiene como finalidad crear un objeto con una serie de atributos que intermedia entre nuestra capa de persistencia (nuestras entities) y lo que pueda ser enviado o recuperado por el cliente del servidor, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple. O por ejemplo, imaginemos la situación que no queramos que la información de determinados atributos de las entidades sean visibles por los clientes. Si ponemos una clase DTO en medio (los clientes lo que ven son los DTO) el problema se resuelve (eso podría solucionarse también con alternativas como @JsonIgnore) .

Ejemplo, tenemos una Entity Usuario que tiene como atributo password. Ese dato, probablemente, no queramos que sea mostrado a un cliente que pregunte por la tabla usuarios. Cuando nosotros nos creamos nuestro UsuarioDTO tendríamos algo así:

```
public class UsuarioDTO{

    private Integer idusuario;

    private String nombre;

    private String password;

    public UsuarioDTO() {}


    //no se muestran los getter y setter por ser los habituales
}

… // aquí más adelante en el código. En un método donde devolvemos al
//cliente la información de un objeto usuario:
```

Otro motivo importante para usar un DTO es si hay aplicaciones que están montadas sobre nuestro servicio REST ¿ Qué ocurre si nosotros modificamos nuestras Entities ? Automáticamente las aplicaciones de usuario que hagan uso de nuestro servicio fallarán porque no tienen contemplados los cambios. Si tenemos las clases DTO, en todo caso, lo único que tendremos que modificar es el mapeo entre nuestras entities modificadas y las clases DTO. De cara a las aplicaciones de usuario que usen nuestro servicio será transparente porque seguirán viendo las DTO como ya estaban

Crearemos un paquete: dto

Veamos un ejemplo de clase allí (La clase está pensada para Usuario en la app de tienda. La idea es que un usuario de nuestra app puede tener varios roles y cada rol lo pueden tener varios usuarios. No se ponen los getter y los setter para mejor visión)

```
public class UsuarioPostDTO {
    private static final long serialVersionUID = 1L;
    private Integer idusuario;
    private String nombre;
    private String password;
    private Collection<String> roles;

     public UsuarioPostDTO(Usuario u) {
          super();
          this.idusuario = u.getIdusuario();
          this.nombre = u.getNombre();
          this.password = null;
          ArrayList<String> roles = new ArrayList<String>();
          for(UsuarioRol ur : u.getUsuarioRolCollection() ) {
               roles.add(ur.getFkRol().getNombre());
          }
          this.roles = roles;
     }

     public UsuarioPostDTO() {
          super();
     }
}
```

Observar que en este ejemplo (no será así como lo tendremos en nuestra aplicación final) se ha elegido “puentear” las tablas intermedias y hay una lista de String con los roles que va a tener el usuario. De esta forma mediante los objetos DTO simplificamos la visión que tienen los clientes de nuestra aplicación (con las ventajas e inconvenientes que eso implica)

Las claves se devolverán como datos nulos.

Ahora lo que nos faltaría es activar una clase controlador para que reaccione a una request del cliente web. Vamos a crear ahora el primer controlador que se encargará de alguna de las rutas REST

Práctica 7: Crear las clases DTO pertinentes. Como sabemos, las DTO son las que va a ver el usuario y las que les vamos a mostrar el JSON. Anotar con @JsonIgnore allí donde se precise (Por ejemplo, si una monedaDTO tiene una colección de historicoDTO entonces hay que impedir los bucles ya que a su vez cada histórico tiene una moneda. Podremos poner un @jsonignore en el atributo moneda de la clase HistoricoDTO)

Creación de Controlador y manejo de rutas REST en Spring

Supongamos que queremos gestionar los Producto de la aplicación tienda. Veamos un fichero controlador (lo pondremos en el paquete controller)

Ejemplo de petición GET en REST Controller

```
@RestController
@CrossOrigin
@RequestMapping("/api/v1/monedas")
public class MonedasREST {

	@Autowired
	MonedasService monedasService;

	@GetMapping //si queremos subruta lleva paréntesis
	public ResponseEntity<?> getAll(){
		List<MonedaDTO> monedasdto = null;

		Iterable<Moneda> findAll = monedasService.findAll();
		if(findAll != null) {
			monedasdto = new ArrayList<MonedaDTO>();

			for (Moneda m : findAll) {
				MonedaDTO mdto = new MonedaDTO();
				mdto.setNombre(m.getNombre());
				mdto.setPais(m.getPais());
				mdto.setIdmoneda(m.getIdmoneda());
				if( m.getHistoricocambioeuros() != null ) {
					mdto.setHistoricos(new ArrayList<HistoricoDTO>());
					for (Historicocambioeuro h:
m.getHistoricocambioeuros()) {
						HistoricoDTO hdto = new HistoricoDTO();

	hdto.setIdhistorico(h.getIdhistoricocambioeuro());
						hdto.setFecha(h.getFecha());

	hdto.setEquivalenteeuro(h.getEquivalenteeuro());
						hdto.setMonedadto(mdto); /

						mdto.getHistoricos().add(hdto);
					}
				}

				monedasdto.add(mdto);
			}
```

```
        }
        return ResponseEntity.ok(monedasdto);
```

@RestController nos trata la clase como un controlador REST

`@CrossOrigin` nos evita problemas de prohibición de cors cuando una aplicación cliente web (por ejemplo react) nos consulta la api

@RequestMapping nos dice que vamos a gestionar las rutas /api/v1/monedas en solicitudes de monedas (el controlador se desencadenará cuando el cliente haga una request que empiece por: /api/v1/monedas)

Vemos que hemos inyectado MonedaService porque hemos puesto @Autowired

@GetMapping Existe GetMapping, PostMapping,… le estamos diciendo que nos mapée las solicitudes (request) de usuario con el método al que está anotando. En este caso queremos trabajar las solicitudes de tipo GET a las rutas: /api/v1/monedas (**observar que NO SE HA PUESTO SUBRUTA ASÍ QUE TOMA LA RUTA GENERAL ESPECIFICADA EN LA CLASE)** Como la clase ya está respondiendo a: /api/v1/monedas no tenemos que poner la ruta completa en el mapping.

Si quisiéramos manejar subrutas sería así:

@GetMapping("subruta") En el caso anterior estaría respondiendo a las rutas:

/api/productos/subruta

Basta con poner: “subruta” y se entiende que es una subruta de su padre: /api/productos Luego nos queda: /api/productos/subruta

Si hubiéramos querido responder a solicitudes de tipo POST habríamos puesto: PostMapping

ResponseEntity.ok() lo usuamos cuando todo ha ido bien. ResponseEntity permite devolver mensajes de estado diferentes (errores por ejemplo)

Observar también que hemos puesto un logger, para hacer seguimiento de errores. Es importante establecer en las propiedades del proyecto (application.properties) el nivel de logger para el paquete:

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/almacentienda?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=1q2w3e4r
logging.level.org.hibernate.SQL  =debug
logging.level.es.iespuertodelacruz.jc.monedas= DEBUG
spring.jpa.show-sql=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

Apuntar la importancia de nombres significativos en nuestras clases, para una clase que sea controlador de los productos, por ejemplo: ProductController, o ProductREST para especificar que es un controlador para generar recursos REST

Ejemplo de petición *DELETE* en REST Controller

Veamos un ejemplo en la aplicación de monedas respecto al borrado (http delete)

```
	@DeleteMapping("/{id}")
	public ResponseEntity<?> delete(@PathVariable Integer id){
		Optional<Monedas> optM = monedasService.findById(id);
		if(optM.isPresent()) {
			monedasService.deleteById(id);
			return ResponseEntity.ok("moneda borrada");
		}else {
			return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("el id del registro no
existe");
		}
	}
```

@PathVariable es importante para que nos inyecte la parte: {id} en nuestra variable:

Integer id. Gracias a las llaves: {} Spring sabe que es variable

En esta ocasión vemos dos ResponseEntity. En un caso informando de consulta mala

Ejemplo de petición POST en REST Controller

Vamos a ver ahora un POST (este método, como otros, se modificará al usar las clases DTO, pero por ahora nos vale)

```
	@PostMapping
	public ResponseEntity<?> save(@RequestBody Monedas mDTO){
		Monedas m = new Monedas();
		m.setNombre(mDTO.getNombre());
		m.setPais(mDTO.getPais());
		monedasService.save(m);

		return ResponseEntity.ok().body(new Monedas(m));
	}
```

Vemos que no tiene mucho sentido hacer los setter con una nueva moneda. Podríamos realizar el save directamente. Ahora bien, cuando tengamos clases DTO sí que tiene sentido.

Ejemplo de petición PUT en REST Controller

```
	@PutMapping("/{id}")
	public ResponseEntity<?> update(@PathVariable Integer id,
@RequestBody Monedas mDTO){
		Optional<Monedas> optM = monedasService.findById(id);
		if(optM.isPresent()) {
			Monedas m = optM.get();
			m.setNombre(mDTO.getNombre());
			m.setPais(mDTO.getPais());
			return ResponseEntity.ok(monedasService.save(m));
		}else {
			return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("el id del registro no
existe");
		}
	}
```

Práctica 8: Crear todos los métodos necesarios para la api funcional (PUT, DELETE, etc)

Realizando lo detallado para nuestras dos tablas (entities, repositorios, servicios, controladoresrest) ya tendríamos un servicio REST operativo… muy elemental y sin seguridad, pero ya es operativo

Consultas REST y Filtrar, Ordenar,...

Imaginemos que queremos filtrar la salida de la tabla de usuarios de tal forma que el nombre del usuario contenga un texto que le pasemos. Una posible alternativa podrían ser consultas:

GET /api/usuarios?nombre=juan

La anterior consulta nos devolvería los usuarios con nombre juan

Para lo anterior necesitamos recibir parámetros que no son de path (como lo era el ID)

Veamos ejemplo (pensado en una app de tienda y el caso de usuarios con varios roles posibles):

```
@RestController
@CrossOrigin
@RequestMapping("/api/usuarios")
public class UsuarioREST {
	private static final Logger logger =
LoggerFactory.getLogger(UsuarioREST.class);
	@Autowired UsuarioService usuarioService;
	@Autowired RolService rolService;
	@Autowired UsuarioRolService usuarioRolService;

	@GetMapping
	public ResponseEntity<List<UsuarioPostDTO>> getAll(
		@RequestParam(required=false, name="nombre") String strNombre
	){
…
}
```

@RequestParam va a inyectar el parámetro llamado: nombre en la String strNombre en el caso que lo hayan enviado (required=false)

Ahora se estará respondiendo a las consultas: GET /api/usuarios?nombre=juan

Simplemente discriminaremos el arraylist que construimos con la respuesta ya sea a nivel de DDBB (con la query pertinente) o a al ir agregando elementos al arraylist desde Java.

A nivel de Java es tan sencillo como:

```
usuarioService
     .findAll()
     .forEach(p -> {
          Usuario u = (Usuario)p;
          if( strNombre == null || (strNombre != null &&
u.getNombre().contains(strNombre))) {
               UsuarioPostDTO uDTO = new UsuarioPostDTO(u);
               prods.add( uDTO);
          }

     } );
```

Como se ve, únicamente se agregan a la lista los objetos con el nombre correctos

El problema a nivel de Java es que nunca es tan eficiente como consulta where a la DDBB. Vamos a ver como se hace a nivel de DDBB y así aprendemos a hacer queries personalizadas en Spring

Para eso tendremos que agregar métodos nuevos a nuestros repositorios

Agregar Queries nuevas a los Repositorios

Hasta ahora habíamos visto que nuestros repositorios hacían poca cosa… simplemente extendían de JpaRepository y ya Spring nos inyectaba todo lo que necesitamos.

Veamos un ejemplo de Repositorio con query agregada:

```
public interface UsuarioRepository extends JpaRepository<Usuario,
Integer> {
    //@Modifying
    @Query("SELECT t FROM Usuario t where t.nombre = :name")
    List<Usuario> findByNombre(@Param("name") String strNombre);
}
```

Nota: si queremos modificar la DDBB (no lectura) hay que poner: **@Modifying**

Nosotros definimos el prototipo del método (findByNombre) Inyectando mediante:

`@Param`(“`name`”) String strNombre

el parámetro “`name`” que precisa la query. El valor del String: `strNombre` es el que se le pasará como parámetro a la query (observar que “`name`” en el trozo de código está del mismo color para ilustrar ese mapeo)

La query no es diferente de otras en HQL lo único que la hemos tenido que anotar con: @Query para que Spring nos la construya

Una vez creado en el repositorio hay que poner el elemento correspondiente en el servicio: UsuarioService

```
    @Transactional(readOnly=true)
    public List<Usuario> findByNombre(String nom) {
        return
((UsuarioRepository)entityRepository).findByNombre(nom);

    }
```

Por último comentar algo que habíamos nombrado en teoría que son las consultas cruzadas.

Nosotros hemos elegido en la aplicación exponer directamente los recursos (se accede directamente a historicocambioeuro aunque sea una tabla dependiente de la relación 1:N con monedas)

Si hubiéramos elegido la forma de acceder a los recursos: /monedas/1/historicocambioeuro/3 tenemos que poner en nuestra anotación mapping dos variables de path. Vamos a ver un ejemplo

```
	@GetMapping("/{id1}/historicocambioeuro/{id2}")
	public ResponseEntity<?> getHistoricoByMoneda(
				@PathVariable("id1") Integer id1,
				@PathVariable("id2") Integer id2
				){
		Optional<Monedas> optMonedas = monedasService.findById(id1);
		Optional<Historicocambioeuro> optH = optMonedas.get()
				.getHistoricocambioeuroCollection()
				.stream()
				.filter(h->h.getIdhistoricocambioeuro() == id2)
				.findFirst();

		if( optH.isPresent()) {
				return ResponseEntity.ok(
					optH.get()
				);

		}else
				return ResponseEntity.notFound().build();

	}
```

> **Práctica 9**: Hacer endpoint que incluyan búsquedas por parte de un nombre. Por ejemplo, /api/v1/monedas?nombre=a
> devolverá todas las monedas que incluyan la letra a en su nombre

También podemos hacer consultas sql nativas diciéndole a spring que lo es mediante el parámetro: nativeQuery = true

El siguiente ejemplo nos busca los productos por subnombre y precio> 1000:

```
//@Modifying
@Query(
    value= "SELECT * FROM productos WHERE nombre LIKE :name AND precio > 1000",
    nativeQuery = true
 )
 List<Producto> findByNombrePrecio(@Param("name") String strNombre);
```

Nota: si queremos modificar la DDBB (no lectura) hay que poner: **@Modifying**

Hacer test con H2 y el resto con mysql

Agregamos en el paquete config una clase: DatabaseConfig.java

```
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration
public class DatabaseConfig {

    @Bean
    @Profile("!test")
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/gente?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC");
        dataSource.setUsername("root");
        dataSource.setPassword("1q2w3e4r");
        return dataSource;

    }

    @Bean
    @Profile("test")
    public DataSource dataSourceTest() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:file:/tmp/gentetest;DB_CLOSE_DELAY=-1");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;

    }
}
```

Reemplazaremos el nombre de la DDBB, el usuario y la contraseña y el nombre del fichero H2 que estamos poniendo en /tmp (todos los campos marcados en amarillo)

Creamos la carpeta src/test/resources y dentro ponemos el fichero que queramos de creación de nuestra DDBB de pruebas:

Veamos un ejemplo del fichero: src/test/resources/almacentienda.sql:

```
SET MODE MYSQL;

DROP TABLE IF EXISTS productos;
CREATE TABLE productos(
      id INT AUTO_INCREMENT,
    nombre VARCHAR(50),
    precio DECIMAL(7,2),
    stock INT,
    CONSTRAINT pk_productos PRIMARY KEY(id),
    CONSTRAINT uq_nombre UNIQUE KEY(nombre)
);

INSERT INTO productos( id, nombre, precio, stock ) VALUES
(null,"tarjeta ati",120,30),
(null,"pantalla BQ 35",450,21);
```

Finalmente realizamos los test. Podemos aprovechar la clase que nos crea Spring dentro de /src/test/java Un ejemplo

```
@SpringBootTest
@ActiveProfiles("test")
@TestMethodOrder(value = MethodOrderer.OrderAnnotation.class)
@Sql(scripts = {"/almacentienda.sql"})
class GenteApplicationTests {


      @Autowired
      ProductoService productoRepository;


      @Test
      @Order(2)
      void testGetAll() {
            System.out.println("GetAll");
            Iterable<Producto> findAll = productoRepository.findAll();
            ArrayList<Producto> productos = new ArrayList<>();
            for (Producto producto : findAll) {
                  productos.add(producto);
            }
            assertNotNull(findAll);
            assertTrue(productos.size() == 2);

      }
}
```

Subir y bajar ficheros de la API

Hay varios abordajes. Nos vamos a poner en un caso práctico para entenderlo:

Supongamos que tenemos una aplicación cliente que hace un CRUD de productos, donde cada producto tiene: nombre, stock, precio, foto

Cuando queremos crear un nuevo producto desde el cliente le debemos enviar a la api esos 4 datos. Podemos enviarlos todos a la vez (por ejemplo codificado todos los datos y enviados en un único json) y que la api los reciba y guarde el objeto completo en la base de datos

Ese abordaje implica guardar imágenes en la base de datos, lo cual simplifica las copias de seguridad y es más sencillo escalar la api. Ahora bien, por contra hace que la base de datos sea de un tamaño más grande perdiendo eficiencia. Con lo cuál hay casos en los que es una buena alternativa y otros en los que no es tan recomendable

Otra alternativa es enviar de nuevo desde el cliente todos los datos en una única petición (un único json) y que luego la api guarde por un lado la imagen en el sistema de ficheros y por otro lado guarde en la base de datos la ruta a la imagen. En ese caso la dificultad está en contener datos pesados como son las imágenes, en un único json (con la conversión pertinente a base64 y el incremento de más de un 30% del tamaño de la imagen) que luego hay que procesar en el servidor decodificando

Finalmente otra opción es la de que el cliente ejecute dos request: En la primera envia la imagen para guardarla en el sistema de ficheros del servidor y el servidor le devuelve el nombre que va a tener (observar que se enviará en el formato original de la imagen, sin codificar). Luego el cliente hace una segunda petición enviando en un json de producto la ruta de la imagen para que directamente se guarde en la base de datos esa ruta con el resto de datos del producto. La ventaja de este abordaje es que permite separar el manejo de ficheros (subida y bajada) del resto, siendo especialmente lógico si la imagen puede ser reutilizada entre varios usuarios

Por lo anterior nosotros vamos a ver como subir y bajar un fichero de la api de forma independiente al resto de datos de la tabla productos y también vamos a ver como subir en una única petición todos los datos de producto, incluyendo el fichero de imagen (codificaremos la imagen en base64 y se enviará junto con el resto del json)

Vamos a crear un servicio que nos ayude con la tarea. Y luego lo inyectaremos en el controlador.

El servicio tiene un método: get(String filename) para devolver un: resource spring

Este método tomará el fichero almacenado en el sistema de ficheros con el nombre dado y lo devolverá como un recurso, que ya sea un navegador o un cliente javascript (react, angular,…) podrá procesar

Dos métodos save() para guardar el fichero enviado por nuestros clientes en el sistema de ficheros. Uno de los save() recibe un MultipartFile, que responderá al caso de que el fichero se suba directamente, sin conversión a base64

El segundo save() recibe una string (nombre del fichero a guardar) y un array de bytes: byte[] Que es lo que obtenemos después de decodificar la string en base64 recibida por el json de producto

Finalmente tiene un último método llamado: getFilenameFree() que se utiliza para encontrar un nombre de fichero alternativo, en el caso de que haya colisión entre el nombre del fichero recibido con alguno que ya tengamos almacenado

La clase se llama: FileStorageService y la ponemos en el package service:

```
@Service
public class FileStorageService {

  private final Path root = Paths.get("uploads");

  private Path getFilenameFree(String filename){
        Path pathCompleto = this.root.resolve(filename);
        String nombre="";
        String extension = "";
        if( filename.contains(".")) {
              extension = filename.substring(filename.lastIndexOf(".") + 1);
              nombre = filename.substring(0, filename.length() -
extension.length() -1);

        }
        else {
              nombre = filename;
        }
        int contador=1;
        while(Files.exists(pathCompleto)) {
              String nuevoNombre = nombre + "_"+contador;
              nuevoNombre += "."+extension;
              pathCompleto = this.root.resolve(nuevoNombre);
              contador++;
        }
        return(pathCompleto);
}



  public String save(String nombrefichero, byte[] dataFile) {
          //creamos el directorio si no existe
            try {
              Files.createDirectories(root);
            } catch (IOException e) {
              throw new RuntimeException("no se puede crear el directorio");
            }


          try {
            Path filenameFree = getFilenameFree(nombrefichero);
            Files.write(filenameFree,dataFile);
            return  filenameFree.getFileName().toString();
          } catch (Exception e) {
            if (e instanceof FileAlreadyExistsException) {
              throw new RuntimeException("A file of that name already exists.");

            }

            throw new RuntimeException(e.getMessage());

          }

        }
```

```
  public String save(MultipartFile file) {
    //creamos el directorio si no existe
      try {
        Files.createDirectories(root);
      } catch (IOException e) {
        throw new RuntimeException("no se puede crear el directorio");
    }

    try {
      Path filenameFree = getFilenameFree(file.getOriginalFilename());
      Files.copy(file.getInputStream(),filenameFree);
      return  filenameFree.getFileName().toString();
    } catch (Exception e) {
      if (e instanceof FileAlreadyExistsException) {
        throw new RuntimeException("ya existe un fichero llamado así");

      }

      throw new RuntimeException(e.getMessage());

    }

  }


  public Resource get(String filename) {
    try {
      //obtenemos la ruta al fichero en nuestra carpeta, dado el nombre como
parámetro
      Path pathForFilename = root.resolve(filename);

      //queremos devolver un recurso fichero. Obtenemos un recurso para el path
del fichero deseado
      Resource resource = new UrlResource(pathForFilename.toUri());
      if( resource.exists()) {
        return resource;
      }else {
        throw new RuntimeException("no se puede acceder a " + filename);
      }
    } catch (MalformedURLException e) {
      throw new RuntimeException("Error: " + e.getMessage());
    }
  }

}
```

Veamos ahora el controller

Ejemplo de subir un fichero en su formato original:

```
      @PostMapping("/upload")
        public ResponseEntity<?> uploadFile(@RequestParam("file") MultipartFile
file) {
          String message = "";
          try {
            String namefile = storageService.save(file);

            message = "" + namefile;
            return ResponseEntity.status(HttpStatus.OK).body(message);
          } catch (Exception e) {
            message = "Could not upload the file: " + file.getOriginalFilename()
+ ". Error: " + e.getMessage();
            return
ResponseEntity.status(HttpStatus.EXPECTATION_FAILED).body(message);
          }
        }
```

Ejemplo para subir un producto completo con la imagen (lo que recibimos lo llamamos ProductoDTO) La clase producto tiene un atributo foto que es un string con la ruta donde está almacenado

```
      @PostMapping
        public ResponseEntity<?> nuevoProducto(@RequestBody ProductoDTO
productoDTO) {
            Producto producto = new Producto();
            producto.setId(productoDTO.getId());
            producto.setNombre(productoDTO.getNombre());
            producto.setPrecio(productoDTO.getPrecio());
            producto.setStock(productoDTO.getStock());


            String codedfoto = productoDTO.getFotoBase64();
            byte[] photoBytes = Base64.getDecoder().decode(codedfoto);

            String nombreNuevoFichero =
storageService.save(productoDTO.getFotoNombre(),photoBytes);

            producto.setFoto(nombreNuevoFichero);
            Producto save = productoService.save(producto);
            return ResponseEntity.ok(save);
        }
```

Ejemplo de un endpoint que permite descargar el fichero en su formato original:

```
        @GetMapping("/ficheros/{filename}")
        public ResponseEntity<?> getFiles(@PathVariable String filename) {
          Resource resource = storageService.get(filename);



          // Try to determine file's content type
          String contentType = null;
          try {
            contentType =
URLConnection.guessContentTypeFromStream(resource.getInputStream());

          } catch (IOException ex) {
              System.out.println("Could not determine file type.");
          }

          // Fallback to the default content type if type could not be
determined
          if (contentType == null) {
              contentType = "application/octet-stream";
          }

          String headerValue = "attachment; filename=\"" +
resource.getFilename() + "\"";

          return ResponseEntity.ok()
                  .contentType(MediaType.parseMediaType(contentType))
                  .header(

      org.springframework.http.HttpHeaders.CONTENT_DISPOSITION,
                              headerValue
                  )
                  .body(resource);
        }
```

Vemos un ejemplo de subir el fichero en formato original con React:

```
function UploadFichero() {
const [mensajes, setmensajes] = useState("");

async function subirfichero(ev: FormEvent<HTMLFormElement>){
  ev.preventDefault();
  let formulario = ev.currentTarget;
  let file = formulario.inputfichero.files[0];

  if (file) {
    const formData = new FormData();
    formData.append("file", file);
    try {
      let response = await axios.post('http://localhost:8080/api/v1/upload',
        formData, {
          headers: { 'Content-Type': 'multipart/form-data' }
        }
      );
      let respuesta = "";

      if(response.data){ respuesta = JSON.stringify(response.data);  }

      setmensajes(respuesta);
    } catch (error) {    console.log("error dice: "+error);      }
 }
}

return (
 <div >
 <form onSubmit={subirfichero}>
 <label htmlFor="file" >
 elegir fichero
 </label>
 <input id="inputfichero" type="file" />
 <button type="submit">Subir</button>
 </form>

 mensajes: {mensajes}
 </div>
);
}
```

Y un ejemplo, también en React, que sube producto completo con el fichero en Base64:

```
interface Producto {
  id?:     number;
  nombre: string;
  precio: number;
  stock:  number;
  fotoBase64?: string;
  fotoNombre?: string
}

const SubirProductoConImagen = (props: Props) => {
  const [file, setfile] = useState<File | null>(null);
  const [nombrefichero,setnombrefichero ] = useState("");

  const [mensajes, setmensajes] = useState("inicio");


  const [photoBase64, setphotoBase64] = useState("");

  function handleFileChange(evt: React.ChangeEvent<HTMLInputElement>) {
    evt.preventDefault();
    let newFiles = evt.currentTarget.files ?? [];
    if (newFiles[0]) {
      setfile(newFiles[0]);
    }

  }




  async function crearproductoCompleto(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    let formulario = event.currentTarget;
    const nombre = formulario.nombre.value?? "prod01";
    const precio = Number(formulario.precio.value??0);

    let dataBase64 = photoBase64;

    dataBase64 = dataBase64.replace(/^.*base64,/, "");
```

```
    let product:Producto ={
      nombre,
      precio,
      stock: 200,
      fotoBase64: dataBase64,
      fotoNombre: nombrefichero
    }



    try {
      const response = await
axios.post('http://localhost:8080/api/v1/productos', product);
      console.log(response.data);
    } catch (error) {      console.error(error);    }

  }

  return (
    <div className="App">

        <form onSubmit={crearproductoCompleto}>
              <div>
                <label htmlFor="nombre">Name:</label>
                <input
                  type="text"
                  id="nombre"

                />
              </div>
              <div>
                <label htmlFor="precio">precio:</label>
                <input
                  id="precio"
                  type="text"
                />
              </div>
              <div>
                <label htmlFor="photo">Photo:</label>
                <input
                  type="file"
                  id="photo"

                  onChange={(event) => {
                    if(event.currentTarget.files){
                      const file = event.currentTarget.files[0];
```

```
                    const fileReader = new FileReader();

                    fileReader.readAsDataURL(file);

                    fileReader.onload = () => {
                      setnombrefichero(file.name);
                      setphotoBase64(fileReader.result as string);
                    };
                  }
                }}
              />
            </div>
            <button type="submit">Submit</button>
        </form>
    </div>
  );
}
```

Securizando la API REST

Por la forma que tienen los servicios REST que se acomodan perfectamente a un protocolo sin conexión como es HTTP , ¿ debemos abordar la seguridad de estos servicios mediante sesiones de servidor ?

Tenemos que entender bien que HTTP es inherentemente NO ORIENTADO A CONEXIÓN. Para las aplicaciones web se ha usado casi siempre el concepto de sesión HTTP para enfrentarse a esa situación. ¿ Cómo lo hace ? Cuando el usuario se autentica contra el servidor se genera un ID único y se envía al cliente. Cada vez que el cliente haga algo contra el servidor envía ese ID de sesión (típicamente mediante una cookie) . Esta sesión tiene habitualmente un tiempo de vida limitado ¿ por qué no es una alternativa válida para un servicio REST ?

REST como hemos dicho antes pretende utilizar las cualidades del protocolo HTTP, que no está pensado para trabajar orientado a conexión. Adicionalmente las sesiones tienen problemas (Por motivos de seguridad: cuanto más tiempo más fácil que se pueda quebrar. Por motivos de optimización de recursos: Toda la info de sesión tiene que estar preparada y accesible en todo momento para la sesión. Por escalabilidad: ¿ cómo hacemos para conservar las sesiones de servidor si una consulta va a un servidor y la siguiente consulta la resuelve otro de nuestros servidores? <nota: hay solución, por supuesto, pero es un esfuerzo innecesario si no se usan sesiones>). Siendo así ¿ qué otras alternativas tenemos para la autenticación si no se usan sesiones ?

- Basic auth

En Basic auth el cliente SIEMPRE envía user/pass. Esta información se almacena en la cabecera del mensaje. Para enviarla sin problemas por HTTP (recordar los problemas si estuvieramos escribiendo en UTF al enviar por HTTP) se codifica en base64 **¡esto no es para nada encriptar!** Imaginemos que tenemos la información en la siguiente string:

```
username:password
```

lo anterior quedaría codificado en base64 así: dXNIcm5hbWU6cGFz

Entonces la cabecera HTTP diría:

```
Authorization: Basic dXNIcm5hbWU6cGFz
```

Esto tiene muchos problemas de seguridad. Se recomienda que si se elige sea con HTTPS

-

Digest

Como en el basic auth, se envía usuario y contraseña en cada solicitud. La diferencia es usar función hash para usuario/pass, ejemplo, si usamos md5: **MD5(username:realm:password)**. Aún siendo mejor que la anterior se debe usar con HTTPS. Observar vulnerabilidad man-in-the-middle: si una falsificación de nuestro servicio solicita basic auth y luego nos envía a nosotros digest ni el cliente ni nuestro servidor lo sabrá. Por ello se recomienda HTTPS

- Oauth

A modo de resumen, básicamente hay un tercero de confianza para las dos partes que **da un código de un único uso** al cliente (una vez el cliente se ha autenticado en facebook, o algún otro que provea el servicio Oauth) para que nuestro servidor sepa que el cliente es quién dice ser. Nuestro servidor contacta con facebook aportando el id del cliente, el código de único uso y nuestra propia clave de autorización con facebook. Una vez validado, facebook devolvería un token a nuestro servidor para ser usado por el cliente en cada solicitud que haga a nuestro servidor

- JWT

Json Web Token es de las alternativas más usadas donde generamos un token. Una vez que el usuario se ha autenticado en nuestro servicio mediante password, se le devuelve un token y el usuario usará ese token en cada petición que nos realice

Bien, hemos visto que básicamente o enviamos user/pass en cada petición o usamos un token ¿ y qué es ese token ? Vamos a verlo para el caso de JWT:

JWT

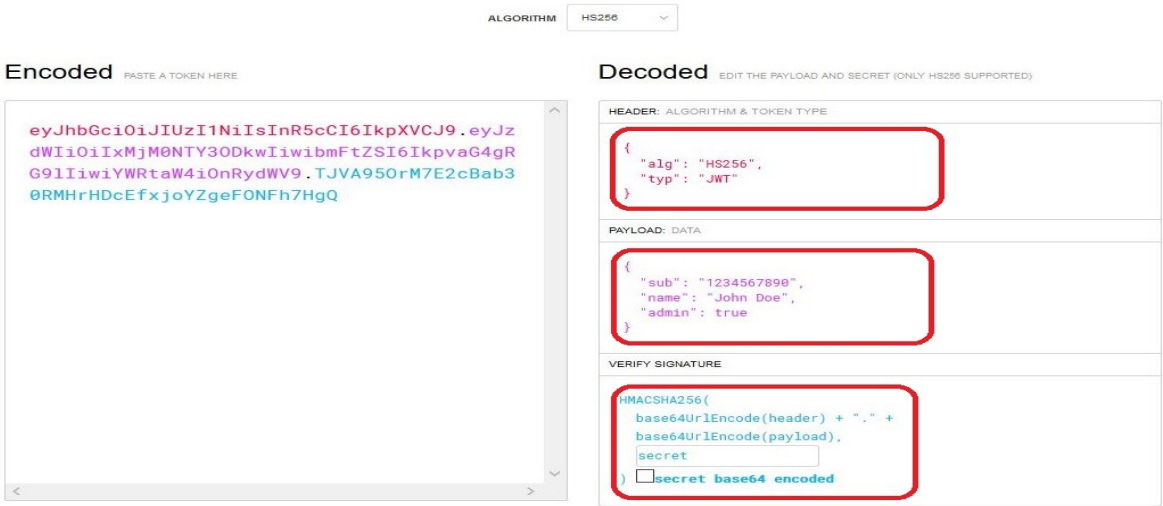
Un JSON Web Token o JWT es un formato estándar, compacto y seguro de trasmitir Claims (propiedades, afirmaciones o en general información) entre diferentes sistemas.

Su gran ventaja es que pueden ser validadas ya que vienen firmadas digitalmente con una clave que se encuentra en el servidor (esto es, los recursos de firmar y validar la firma están siempre y únicamente en el servidor)

Entre otras Claims, podemos enviar desde la IP de la máquina para la que se ha emitido un JWT, los ámbitos a los que se le permite acceder, hasta la fecha y hora de expiración del mismo

Es importante entender que **las Claims no se encriptan**. Es información que se puede leer libremente en el token. Lo que sí se garantiza es que esas claims son reales, ya que la firma les da validez.

Bien, vamos a ver como es un token. Lo siguiente es una muestra que se puede trabajar en: https://jwt.io/ en esa web podemos probar nuestros token

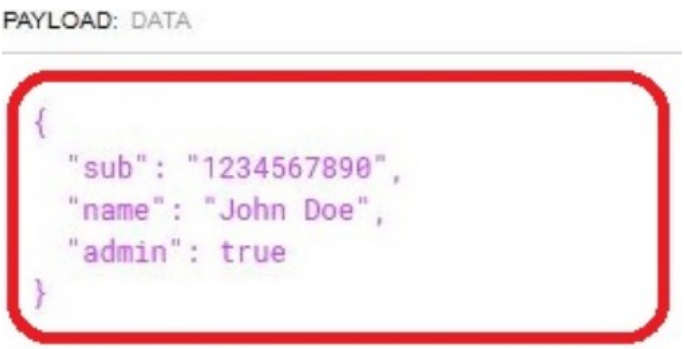


El token va codificado (que no encriptado) y sería lo que vemos en la pantalla de la izquierda. Convenientemente decodificado por la web que hemos nombrado podemos observar 3 partes en el token (diferenciadas por los colores):

cabecera-header.carga-payload.firma-signature

Vemos que en la cabecera se informa que el algoritmo de firma es: HS256 y que estamos con un token de tipo: JWT

En la sección de Payload van los claims. Y observamos que aparece:



sub: por subject: identifica al usuario

Vemos que se han elegido otras claims que no son “estandar” en este caso:

name: para el nombre completo

admin: boolean que informa del rol de usuario

En la sección de firma se codifica la información de header y payload mediante la clave del servidor. Al ser una clave que únicamente conoce el servidor, los datos se pueden verificar como reales en el servidor (ya que puede codificar los datos recibidos con la clave y obtener la misma firma) . Debemos tener claro que con esto se consigue garantizar que los datos son reales (ya que el servidor asegura la firma) pero los datos que van en los claims NO SON ENCRIPTADOS son de libre consulta

Veamos el proceso de firma más detenidamente:

```
key  = 'secretkey'

unsignedToken = encodeBase64Url(header) + '.' + encodeBase64Url(payload)

signature = HMAC-SHA256(key, unsignedToken)
```

‘secretkey’ es la parte que conoce únicamente nuestro servidor.

Vemos que aplicamos la función: HMAC-SHA256(key, unsignedToken) que aplica la clave ‘secretkey’ mediante el algoritmo a los datos que queremos enviar con el token: **header+payload** Obtenemos así un texto cifrado que únicamente se puede obtener si disponemos de la clave del servidor: ‘secretkey’ asegurando así que la información del token es verdadera

Vamos a ver como sería el procedimiento de autenticación mediante token:

- El usuario accede a un login e introduce su username y password

- El servidor valida la información y establece que información podría necesitar para futuras consultas del usuario. Por ejemplo, podría decidir que quiere:

. un claim para el nombre de usuario

. un claim para el rol del usuario

. un claim para establecer el tiempo de expiración del token

Dada esa situación la información de payload podría ser:

```
{
        “sub”:  “ana”,
        “rol”: “usuario”,
        “exp”:  “1437531911”
}
```

- Establece la cabecera del token:

```
{
        “alg”:  “HS256”,
        “typ”: “JWT”
}
```

- Codifica cada uno de esos JSON mediante base64 para que no haya problemas para ser enviado mediante HTTP (debemos tener claro que esto no es encriptación). Para el caso anterior nos quedaría:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.**eyJzdWIiOiJhbmEiLCJyb2wiOiJ1c3VhcmlvIiwiZXhwIjoiMTQzNzUzMTkxMSJ9**

- En el servidor debe haber almacenada una clave (que no se eliminará al menos durante el tiempo de vida del token) para firmar. Esa clave la conoce únicamente el servidor ocurriendo que únicamente con esa clave se consigue la firma para los datos originales (header + payload)

El token resultante sería:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.**eyJzdWIiOiJhbmEiLCJyb2wiOiJ1c3VhcmlvIiwiZXhwIjoiMTQzNzUzMTkxMSJ9**.**ghuunhqBPFEMMrD2Ke3-bznTbghgdDGs4AXThWtioxA**

- El servidor le envía el token al usuario. Éste cada vez que quiera una petición enviará también el token para autenticarse. El token lo envía el usuario en la cabecera: Authorization y suele ir acompañado de la palabra: Bearer

Authorization: **Bearer** **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**. **EyJzdWIiOiJhbmEiLCJyb2wiOiJ1c3VhcmlvIiwiZXhwIjoiMTQzNzUzMTkxMSJ9**. **ghuunhqBPFEMMrD2Ke3-bznTbghgdDGs4AXThWtioxA**

- El servidor recibe la petición con el token. Mira la información del payload:

```
{
        "sub":  "ana",
        "rol": "usuario",
        "exp":  "1437531911"
}
```

Como la fecha aún no se ha alcanzado el token aún es válido y sin tener que consultar las bases de datos sabe que quién le hace la consulta es: “ana” y que tiene por rol: “usuario” . Para evitar el problema de falsificaciones y supuestos usuarios que dicen llamarse “ana” el servidor hace el proceso de firma mediante el algoritmo descrito en la cabecera (HS256) de los datos del token. Observa que el texto obtenido es el mismo que el que ha recibido en la firma y así verifica que la información es verdadera

¿ Hay alguna norma para la forma en la que enviamos el token al cliente ?

No hay un estandar sobre como entregar el token al cliente. Sin embargo podemos observar que si se envía el token como una cabecera al cliente se da la posibilidad de darle una respuesta en el cuerpo del mensaje al cliente. Ahora bien, si estamos trabajando únicamente una Api REST quizás no nos sea tan interesante una respuesta y simplemente enviamos el token en el cuerpo del mensaje. Enviar el token en el cuerpo del mensaje o incluso en una Cookie parece una solución perfectamente válida. Otra cosa es luego como nos envía el cliente al server en subsiguientes consultas. En ese caso se espera que el cliente envíe una cabecera:

Authorization: Bearer

Veamos un ejemplo de mensaje HTTP enviado por el cliente:

```
POST /rsvp?eventId=123 HTTP/1.1
Host: events-organizer.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. EyJzdWIiOiJhbmEiLC
Jyb2wiOiJ1c3VhcmlvIiwiZXhwIjoiMTQzNzUzMTkxMSJ9. ghuunhqBPFEMMrD2Ke3-bznTbghgdDGs
4AXThWtioxA
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/1.0 (KHTML, like Gecko;
Gmail Actions)
rsvpStatus=YES
```

La cabecera anterior se ajusta al modelo:

Authorization: <type> <credentials>

fue introducida por el W3C en HTTP 1.0 y desde entonces se ha usado en muchas ocasiones. Hay muchos web server que soportan diferentes tipos de autenticación. De esa forma enviar únicamente el token no es suficiente para que sepa el servidor el método apropiado a aplicar

En su mayoría los servicios que implementan Oauth 2.0 bearer tokens usan ese formato de cabecera. Si bien en nuestro caso no hacemos uso de Oauth, es una buena práctica, así que presupondremos que el cliente nos envía el token de esa forma

Nosotros podemos emular ese comportamiento del cliente, por ejemplo, en el addon de firefox: Rested veamos un ejemplo:

Vemos que en la cabecera Authorization, justo después de la palabra Bearer ponemos el token.

Agregar dependencias maven para JWT en proyecto Spring

Ya las hemos visto al comienzo del tema:

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>java-jwt</artifactId>
            <version>4.4.0</version>
        </dependency>
```

Bien, para trabajar con los token hemos agregado las librerías. También vamos a agregar a nuestro paquete: security, algunas clases de utilidades que nos pueden ser de ayuda:

Patrón Singleton

La clase: GestorDeJWT sigue el patrón singleton (tiene un constructor privado y se genera una única instancia de la clase para toda la aplicación)

```
MiClaseSingleton{
    private MiClaseSingleton(){}
    public static MiClaseSingleton mcs;
    public static getInstance(){
        if(mcs == null){
            mcs = new MiClaseSingleton();
        }
        return mcs;
    }
}
```

Fijarse que se establece a static las cosas que queremos hacer permanente (una instancia de la propia clase y un método para obtener ese objeto, son en este caso) También mirar que el constructor es private (en este caso se ha puesto el constructor vacío. Eso no es lo habitual. La mayor parte de las veces le pondremos contenido) Al ser un constructor privado UNICAMENTE SE PUEDE GENERAR UNA NUEVA CLASE, desde el método static: getInstance()

El motivo por el que se ha elegido ese patrón es porque la clave de JWT debe ser única para poder cifrar y descifrar correctamente los token Con ese patrón se garantiza que habrá esa única clave (hasta que elijamos cambiarla)

JwtService (crear token, validar token)

Podemos poner este servicio (JwtService.java) en el package security. Quedará inyectado al principio de la aplicación y cuando se ejecute un login lo usaremos para crear el token. Así cuando recibamos un token lo validaremos:

```
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.Claim;

@Service
public class JwtService {

    //@Value("${jwt.secret}")
    private String secret="ungransecreto";

    //@Value("${jwt.expiration}")
    private long expiration=9876543210L;

    public String generateToken(String username, String rol) {
        return JWT.create()
                .withSubject(username)
                .withClaim("role", rol)
                .withExpiresAt(new Date(System.currentTimeMillis() + expiration))
                .sign(Algorithm.HMAC256(secret));
    }


    public  Map<String, String> validateAndGetClaims(String token) {
         Map<String, Claim> claims = JWT.require(Algorithm.HMAC256(secret))
                .build()
                .verify(token)
                .getClaims();

         Map<String,String> infoToken = new HashMap<String,String>();
         infoToken.put("username", claims.get("sub").asString());
         infoToken.put("role", claims.get("role").asString());

         return infoToken;
    }
}
```

Viendo la clase expuesta Observamos que se genera automáticamente una clave secreta (Key) para generar los token. Al seguir el patrón singleton hay una única instancia y una única clave secreta. Esta clase es claramente mejorable para las situaciones de cambio de clave (cada x tiempo se cambia una clave estando “vivas” dos claves durante un tiempo de transición para pasar a la siguiente). El método getClaims() es el que se encarga de validar que un token sea válido (tiempo de validez del token, correctamente firmado con la clave secreta,etc) lanzando excepción si no se cumple Finalmente devuelve los datos relevantes (en nuestro caso el nobre de usuario y los roles)

Auth Service

Este servicio podría pensarse como en un: UsuarioService, pero será un poco diferente y de hecho, se apoyará en un UsuarioService que se le inyecta para que hable con la base de datos y crea/valida usuario.

Adicionalmente se le inyecta el JwtService que creamos antes (para el manejo de los token) y la codificación de las password (se inyecta un servicio PasswordEncoder para que la clave en texto plano se guarde en hash)

Adicionalmente vamos a tener que crear una clase nueva para “mapear” la información de la base de datos de un usuario y que sea tratada por la seguridad de Spring como una clase que herede de UserDetails que es específica para el manejo de la autorización

Lo más interesante está en el método getAuthorities() que observamos que está pensado para devolver una lista de roles diferentes. En nuestro caso únicamente vamos a trabajar con un rol, en otro caso debemos modificar apropiadamente

Nota: para la siguiente parte se presupone que tenemos en la DDBB una tabla para usuarios con un campo rol y hemos generado la entity correspondiente

y el repositorio JPA correspondiente (observar que tiene la búsqueda por nombre)

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Integer> {

	@Query("SELECT u from Usuario u where u.nombre=:nombre")
	public Usuario findByName(@Param("nombre") String nombre);
}
```

Veamos ahora la clase LoginService (también en **package security**)

```
@Service
public class AuthService {


    @Autowired
    private UsuarioRepository usuarioRepository;

    @Autowired
    private JwtService jwtService;

    @Autowired
    private PasswordEncoder passwordEncoder;


       public String register(String username, String password, String email) {
              Usuario usuario = new Usuario();
              usuario.setNombre(username);
              usuario.setPassword(passwordEncoder.encode(password));
              usuario.setCorreo(email);
              usuario.setRol("ROLE_USER");

              Usuario saved = usuarioRepository.save(usuario);

              if( saved != null) {
                     String generateToken = jwtService.generateToken(usuario.getNombre(),
usuario.getRol());
                     return generateToken;
              }else {
                     return null;
              }
       }



       public String register(String username, String password) {
              Usuario usuario = new Usuario();
              usuario.setNombre(username);
              usuario.setPassword(passwordEncoder.encode(password));
              usuario.setRol("ROLE_USER");

              Usuario saved = usuarioRepository.save(usuario);

              if( saved != null) {
                     String generateToken = jwtService.generateToken(usuario.getNombre(),
usuario.getRol());
                     return generateToken;
              }else {
                     return null;
              }
       }

       public String authenticate(String username, String password)  {
              String generateToken = null;
              Usuario usuario = usuarioRepository.findByNombre(username).orElse(null);
```

```
            if (usuario != null) {
                  if (passwordEncoder.matches(password, usuario.getPassword())) {
                        generateToken = jwtService.generateToken(usuario.getNombre(),
usuario.getRol());
                  }
            }


            return generateToken;
      }
}
```

Cuidado!!! observar que en el método: register() se está poniendo: `userentity.setRol("ROLE_USER");`

Esto no tiene por qué ser así, ya que se está obligando a que todos los nuevos usuarios sean de ese rol

Hay un objeto de librerías Spring que inyectamos para nuestra gestión, que es: `PasswordEncoder passwordEncoder;` y nos ayuda para convertir a hash una clave y comparar una clave en texto plano con su hash

Cuando compilemos la aplicación con las clases anteriores, veremos que tenemos un aviso de que falta la inyección de passwordEncoder. Hay una clase de configuración que debemos agregar. La pondremos en el **paquete config** y la llamaremos: ApplicationConfig:

```
@Configuration
public class ApplicationConfig {

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

Como vemos se inyecta al inicio de la aplicación el passwordEncoder() Si se quiere este bean podría estar en el @Configuration de SecurityConfiguration.java

Clase AuthController en package controller:

```
@RestController
@CrossOrigin
public class AuthController {

    @Autowired
    private UsuarioRepository usuarioRepository;
```

```
@Autowired
private MailService mailService;

@Autowired
private AuthService authService;

static class UsuarioLogin{
   public UsuarioLogin() {}
   public String nombre;
   public String password;
          public String getPassword() {
                 return password;
          }
          public void setPassword(String password) {
                 this.password = password;
          }
          public String getNombre() {
                 return nombre;
          }
          public void setNombre(String nombre) {
                 this.nombre = nombre;
          }
}

@PostMapping("/login")
public String login(@RequestBody UsuarioLogin u ) {
    //return "recibe: "+u.nombre + " "+ u.password;
    String token = authService.authenticate(u.getNombre(), u.getPassword());


    if ( token == null ) {
        throw new RuntimeException("Credenciales inválidas");
    }
    return token;

}



static class UsuarioRegister{
   public UsuarioRegister() {}
   public String nombre;
   public String password;
   public String correo;
          public String getPassword() {
                 return password;
          }
          public void setPassword(String password) {
                 this.password = password;
          }
          public String getNombre() {
                 return nombre;
          }
          public void setNombre(String nombre) {
                 this.nombre = nombre;
          }
          public String getCorreo() {
                 return correo;
          }
          public void setCorreo(String correo) {
                 this.correo = correo;
          }
```

```
    }



    @PostMapping("/register")
    public String register(@RequestBody UsuarioRegister u ) {
        //return "recibe: "+u.nombre + " "+ u.password;
        String token = authService.register(u.getNombre(), u.getPassword(),
u.getCorreo());


             String senders[] = {u.getCorreo()};
             mailService.send(senders, "usuario creado", token);
             //return ResponseEntity.ok(token);
        return token;

    }
```

Con lo anterior ya se puede hacer un login y obtener un token

Práctica 10: Crear las clases descritas y obtener un token desde swagger

Supuesto de seguridad de la aplicación (Versiones, roles, filtros etc)

En el paquete security vamos a poner un filtro que analice el token. El objetivo de los filtros en Java es interponerse en los procesos. En este caso en concreto, filtraremos todas las request de usuario de tal forma que los recursos que requieran autenticación no se puedan acceder sin tener un token válido. También veremos que aparte de la autenticación podemos impedir el acceso si no se tiene la autorización pertinente (un usuario válido pero que no tiene rol admin intentando acceder para ver la información de otros usuarios, por ejemplo)

URI Versioning

El versionado de servicios es una práctica por la cual, al producirse un cambio en el API de un servicio (no tiene por qué ser únicamente un API REST), se libera una nueva versión de ese servicio de manera que la versión nueva y la anterior conviven durante un periodo de tiempo.

De esta manera, los clientes se irán migrando a la nueva versión del servicio de manera secuencial. Cuando todos los clientes estén consumiendo la última versión del servicio, se retira la anterior.

Lo anterior es relevante para servicios que se amplíen, vayan mejorando y estando aún en producción, queramos tener el menor impacto negativo sobre nuestros clientes mientras estemos con el cambio

En nuestro caso en concreto usaremos más las versiones como una forma para “separar” de una forma limpia las acciones y la visión de nuestro servicio. Vamos a explicar esto un poco más con el ejemplo que estamos realizando de monedas

Versión v1: Se permite que cualquier visitante (no tiene por qué ser un usuario válido) pueda consultar las monedas y los diferentes tipos de cambio que ha tenido (únicamente consultar, nada de modificar, borrar,..)

Versión v2: Esta versión acceden los usuarios autenticados y pueden:

- Ver y **modificar** los datos de las monedas: **insertar** nuevas monedas, modificarlas, **borrarlas**, así como los diferentes valores de cambio históricos.

Es fácil observar que aquí ya es importante diferenciar los accesos ya que en la versión v1 puedes modificar/crear/borrar, cosa que no pueden en la versión v1

- Ver su propio usuario y modificar su contraseña/nombre (eso significa una variación específica del controlador rest para: GET usuarios/{id})

Versión v3: Versión para administradores:

- Pueden realizar cualquier acción sobre la base de datos (son los únicos que tienen permiso para crear usuarios, asignarles roles, etc)

Lo anterior lo podemos conseguir con versiones de url de tal forma que:

GET /api/v3/usuarios/10

Es un acceso de administrador al usuario con id 10 pudiendo ver la información aún perteneciendo a otra persona

GET /api/v1/monedas

Es un acceso de cualquier persona. No tiene por qué estar autenticado y podrá ver los datos de las monedas que estimemos conveniente

GET /api/v1/usuarios/10

Directamente será un recurso que no existirá en los accesos a visitantes. Ya que no se permite en la versión1 (para visitantes) ver la información de los usuarios

GET /api/v2/usuarios/10

Es un recurso accesible si quién lo consulta es el **usuario con id 10**. El resto de los usuarios no podrá.

Lo anterior significa que un usuario: root que tendrá rol administrador si accede a las versiones v1 y v2 tampoco podrá modificar/ver usuarios Únicamente cuando acceda a la versión v3

Clase JwtFilter en **package security**:

```
@Component
public class JwtFilter extends OncePerRequestFilter {

	public static final String authHeader="Authorization";
	public static final String authHeaderTokenPrefix="Bearer ";

    @Autowired
    private JwtService jwtTokenManager;

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain)
            throws ServletException, IOException {


	String path = request.getRequestURI();


	//rutas permitidas sin estar autenticado
	String rutasPermitidas[]= { "/swagger-ui.html",
                                    "/swagger-ui/", "/v2/",
                                    "configuration/",   "/swagger",
                                    "/webjars/", "/api/login",
                                    "/api/register", "/v3/",
                                    "/websocket", "/index.html", "/api/v1"};

	//String rutasPermitidas[] = {};


	for (String ruta : rutasPermitidas) {
	     if (path.startsWith(ruta)) {
	         // Permitir la solicitud sin autenticación
	         filterChain.doFilter(request, response);
	         return;
	     }
	 }

	//el token viene en un header Authorization
	 String header = request.getHeader(authHeader);

	 //típicamente se precede el token con bearer:  Bearer token
	 if (header != null && header.startsWith(authHeaderTokenPrefix)) {

	     String token = header.substring(authHeaderTokenPrefix.length());
	     try {
	  Map<String, String> mapInfoToken = jwtTokenManager.validateAndGetClaims(token);

	      //System.out.println(mapInfoToken);
```

```
            final String nombreusuario=mapInfoToken.get("username");

            final String rol = mapInfoToken.get("role");


            //UserDetails en Spring Security es un interfaz basado en Principal de java
            //y es la forma que tiene Spring de mantener la información de usuario
"autenticado"
            //en el contexto de seguridad. Nos permite guardar la información de
username
            //y authorities ( los roles si se admiten múltiples roles ) Creamos un
objeto de clase anónima UserDetails:
            UserDetails userDetails = new UserDetails() {

                  String username=nombreusuario;

                  @Override
                  public Collection<? extends GrantedAuthority> getAuthorities() {
                      List<GrantedAuthority> authorities = new ArrayList<>();

                      authorities.add(new SimpleGrantedAuthority(rol));
                      return authorities;
                  }

                  @Override
                  public String getPassword() { return null;    }

                  @Override
                  public String getUsername() {
                        return username;
                  }

            };



UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                              userDetails,
                              null,
                              userDetails.getAuthorities()
                  );

  authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                  SecurityContextHolder.getContext().setAuthentication(authToken);



                  filterChain.doFilter(request, response);

            } catch (JWTVerificationException e) {
                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                return;
            }

        }
    }
}
```

La anterior clase toma el token (cuidado! Tiene en cuenta que lleva la palabra: Bearer justo antes del token) y lo valida. Después toma la información de nombre de usuario y rol para ponerlo en el contexto de seguridad de Spring y así poder aplicar políticas de autorización

Para inyectar el filtro, detallar las rutas donde se aplica autenticación y en cuales también autorización haremos uso de un último fichero de configuración: **SecurityConfiguration** en **package: config**

Nota: Hay que desactivar el fichero de configuración: ApplicationNoSecurity porque hace justo lo contrario (quita toda seguridad a las rutas). Podemos hacerlo, tanto borrándolo como simplemente poniendo comentarios en: //@Configuration y así ya no lo carga como un fichero de configuración

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {



	@Autowired	private JwtFilter jwtAuthFilter;

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

	@Bean
	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{

		http
			.cors(cors->cors.disable())
			.csrf(csrf -> csrf.disable() )
				.authorizeHttpRequests(auth -> auth
					.requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()


					.requestMatchers(
					"/", "/swagger-ui.html",
					"/swagger-ui/**", "/v2/**",
					"/configuration/**",	"/swagger*/**",
					"/webjars/**", "/api/login",
					"/api/register", "/v3/**",
					"/websocket*/**", "/index.html", "/api/v1/**"
					).permitAll()

					.requestMatchers("/api/v3/**").hasRole("ADMIN")
					.anyRequest().authenticated()
				)
				.sessionManagement(sess ->
sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
				.addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);
```

```
            return http.getOrBuild();
        }
```

Lo que podemos destacar más son los antmatches que están dejando sin controlar todas las rutas de swagger, /api/login y /api/register así como las de v1: /api/v1/**

Veamos el siguiente trozo:

```
.requestMatchers("/api/v3/**").hasRole("ADMIN")
```

Con este fragmento utilizamos expresiones regulares para decirle que cualquier ruta que esté bajo: /api/v3 tiene que ser autorizado (observar authorizeRequests()) por medio de que tenga el rol ADMIN. Algo que es importante es que tengamos en cuenta que Spring considera que los roles van escritos siempre: ROLE_ADMIN, ROLE_USER así que va a agregar lo que nosotros hayamos puesto ahí a: ROLE_

`.anyRequest().authenticated()` con este trozo le estamos indicando que cualquier request debe estar autenticada

Personalizar la respuesta según el usuario que haga la consulta

Hemos visto como filtrar diferentes end-points mediante la aplicación de roles y restringir el acceso a recursos (por ejemplo, únicamente admin para /api/v3) Sin embargo también hay casos que queremos personalizar una respuestas según quién haga la consulta

Imaginemos que queremos permitir a los usuarios que accedan y modifiquen la información de su propio usuario pero a ningún otro.

Así por ejemplo, en /api/v1 no habría ninguna información de usuarios

En /api/v2/usuarios se permite ver los nombres de los otros usuarios (recordar que en v2 únicamente estaban accediendo otros usuarios logueados)

Pero para el endpoint:

/api/v2/usuarios/21

nadie debiera tener acceso salvo al nick y la dirección de correo (y únicamente a leer no a modificar) mientras que si eres el usuario 23 sí que debes tener acceso completo.

Para conseguir ese comportamiento haremos uso del contexto de seguridad de spring: **SecurityContextHolder** que tiene dentro la información del usuario que se autenticó con el token:

Así, **en nuestro controller,** para la petición que queramos, tomamos la información del UserDetails principal que nos viene del contexto de seguridad:

```
Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();


            String nombreAutenticado = ((UserDetails)principal).getUsername();

            Usuario u = usuarioService.findById(id);
            if( u.getNombre().equals(nombreAutenticado)) {
                  return ResponseEntity.ok(new UsuarioDTO(u));

            }else {
                  return ResponseEntity
                        .status(HttpStatus.FORBIDDEN)
                        .body("usuario solicitado diferente del autenticado");
            }
```

Fijarse que obtenemos del contexto un objeto UserDetails, que es el motivo por el que nos generamos nuestro propia clase basada en ese interfaz UserDetails, ya que de esa forma podíamos inyectar el objeto UserDetails que ahora estamos recuperando

Enviando correo de verificación al registrar

La dependencia maven:

```
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-mail</artifactId>
            </dependency>
```

Crearemos un fichero de configuración (package config) Se propone el nombre: MailConfig.java:

```
@Configuration
public class MailConfig{

	@Value("${mail.from}") private String mailfrom;
	@Value("${mail.password}") private String mailpassword;

	@Bean
	public JavaMailSender getJavaMailSender(){

		JavaMailSenderImpl sender = new JavaMailSenderImpl();
		sender.setHost("smtp.gmail.com");
		sender.setPort(587);
		sender.setUsername(mailfrom);
		sender.setPassword(mailpassword);

		Properties props = sender.getJavaMailProperties();
		props.put("mail.transport.protocol", "smtp");
		props.put("mail.smtp.auth", "true");
		props.put("mail.smtp.starttls", "true");
		props.put("mail.debug", "true");
		props.put("mail.smtp.starttls.enable", "true");

		return sender;
	}
}
```

Observar la anotación: `@Value("${mail.from}")` Lo que hace es inyectar lo que hayamos definido en resources/application.properties con el nombre: mail.from

Esa es una forma cómoda y centralizada de especificar propiedades. En este caso nos permite establecer la dirección de correo electrónico que envía el correo y su contraseña

El resto de configuración se especificará según sea nuestro proveedor. En este caso el proveedor elegido es google, que usa smtp sobre el puerto 587 (con tls)

Ahora generamos un servicio: MailService.java:

```
@Service
```

```java
public class MailService{
	@Autowired private JavaMailSender sender;

	@Value("${mail.from}") private String mailfrom;


	public void send(String destinatario, String asunto, String  contenido){
		SimpleMailMessage message = new SimpleMailMessage();
		message.setFrom(mailfrom);
		message.setTo(destinatario);
		message.setSubject(asunto);
		message.setText(contenido);

		sender.send(message);

	}

}
```

Nota: si se está usando un servicio de google activar la autenticación en dos pasos y aparecerá un apartado: Contraseñas de aplicaciones. Crear una y copiar la clave generada para usarla en nuestra app

Para hacer el registro correctamente, en nuestra tabla usuarios deberá haber un campo **email** para guardar la información de correo del usuario. También un campo: **activa** que nos ayudará a controlar si el usuario se ha verificado o no. Finalmente un campo: **hash** que será el campo aleatorio que generamos como verificación para que el usuario verifique la cuenta

Una vez el usuario cree el usuario y contraseña habiendo dado su correo, generamos un hash de un número aleatorio grande. El hash lo podemos obtener aplicando sobre el número aleatorio el PasswordEncoder que hemos inyectado como un Bean:
passwordEncoder.encode(numeroaleatorio)

En el correo electrónico enviamos un correo con un enlace a nuestra api. Por ejemplo a un endpoint: registerverify podría ser:

/api/registerverify?usermail=correousuario@correo.es&hash=asjkldjflkasjdflasldk

Bastará con crear un endpoint GET para /api/registerverify con los parámetros usermail y hash

CORS y CSRF

Hemos nombrado un poco de csrf. Vamos a verlo mejor:

CSRF es un ataque: falsificación de petición de sitios cruzados. En el que comandos no autorizados son transmitidos por un usuario (sin éste saberlo) que nuestro sitio confía. Esto ocurre por ejemplo, si un usuario que se ha validado correctamente envía una petición que la página atacante dirige hacia nosotros sin que el usuario lo sepa. Como el usuario está correctamente autenticado, nosotros aceptamos la petición y el ataque tiene lugar. Veamos un ejemplo:

Un ejemplo muy clásico se da cuando un sitio web, llamémoslo "example1.com", posee un sistema de administración de usuarios. En dicho sistema, cuando un administrador se conecta y ejecuta el siguiente REQUEST GET, elimina al usuario de ID: "63": http://example1.com/usuarios/eliminar/63

Una forma de ejecutar la vulnerabilidad CSRF, se daría si otro sitio web, llamemos "example2.com", en su sitio web añade el siguiente código HTML:

<img src="http://example1.com/usuarios/eliminar/63">

Cuando el usuario administrador (conectado en example1.com) navegue por este sitio atacante, su navegador web intentará buscar una imagen en la URL y al realizarse el REQUEST GET hacia esa URL eliminará al usuario 63.

Los frameworks se suelen proteger de éste evente. Como ejemplo en php Laravel, por defecto genera un token por cada usuario del sistema y ese token es el que utiliza el middleware: **VerifyCsrfToken** para verificar si una petición es legítima

Cuando generemos un formulario, se debe incluir un campo: @csrf para que el middleware valide el token:

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Hemos nombrado los middleware ¿ y qué son ?

Los Middleware proveen un mecanismo eficiente para el filtro de peticiones HTTP que ingresen a tu aplicación. Es un puente entre la solicitud y la respuesta que ejecuta un filtrado. Por ejemplo, Laravel incluye un middleware que permite verificar si un usuario está autentificado cuando acceda a tu aplicación. Si el usuario no lo estuviera, el middleware lo redireccionaría a la pantalla de login. Y por el contrario, si lo estuviera, el middleware permitiría el acceso a la aplicación. Es una versión "similar" a los filtros ya que los filtros son middleware también

Si tenemos una aplicación web cliente que se esté conectando con nuestra api (por ejemplo una aplicación angular o react) La aplicación web puede verse como un atacante CORS desde el lado de nuestra api. Y Spring Security por defecto aplica políticas que bloquéan el acceso y responden:

No 'Access-Control-Allow-Origin' .

Ejemplo en angular: Origin **'http://localhost:4200'** no está permitido y se regresa una respuesta con **código de estado 403**

La solución pasa por poner en nuestros controllers la anotación: @Crossorigin

Anexo: mappers, MapStruct

Como sabemos los DTO y las Entities son más un almacén/comunicación de información que pensadas para la lógica de la aplicación. De hecho, en modelos grandes es habitual disponer de las clases del Dominio para que sean las que tienen la lógica de la aplicación y luego hacer transformaciones a DTO y a Entities. Para ello habitualmente usamos mapper que son clases que transforman un objeto en otro.

Mediante MapStruct podemos hacer ese trabajo un poco más automatizado. Debemos agregar a nuestro pom.xml:

```
<dependencies>
          <!-- https://mvnrepository.com/artifact/org.mapstruct/mapstruct -->
        <dependency>
                 <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct</artifactId>
                 <version>1.6.0</version>
        </dependency>


         <dependency>
            <groupId>org.mapstruct</groupId>
            <artifactId>mapstruct-processor</artifactId>
            <version>1.6.0</version>
            <scope>provided</scope>
        </dependency>

</dependencies>

        <build>
        <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>


         <plugin>
             <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-compiler-plugin</artifactId>
             <version>3.13.0</version>
             <configuration>
                 <source>17</source>
                 <target>17</target>
                 <annotationProcessorPaths>
                     <path>
                         <groupId>org.mapstruct</groupId>
                         <artifactId>mapstruct-processor</artifactId>
                         <version>1.6.0</version>
                     </path>
                 </annotationProcessorPaths>
             </configuration>
         </plugin>

        </plugins>
        </build>
```

Creamos un paquete llamado: mappers y creamos un interface para hacer el mapeo. El siguiente ejemplo es para convertir entre la clase Usuario y la clase UsuarioDTO donde UsuarioDTO trata la fechaCreacion como un Date y Usuario lo mantiene como un Long (en la base de datos es un BIGINT)

```
@Mapper
public interface UsuarioMapper {

    UsuarioMapper INSTANCE = Mappers.getMapper(UsuarioMapper.class);


    @Mapping(target = "fechaCreacion", expression = "java(usuario.getFechaCreacion() != null ? new
java.util.Date(usuario.getFechaCreacion()) : null)")
    UsuarioDTO toDTO(Usuario usuario);


    @Mapping(target = "fechaCreacion", expression = "java(usuarioDTO.getFechaCreacion() != null ?
usuarioDTO.getFechaCreacion().getTime() : null)")
    Usuario toEntity(UsuarioDTO usuarioDTO);


    List<UsuarioDTO> toDTOList(List<Usuario> usuarios);


    List<Usuario> toEntityList(List<UsuarioDTO> usuarioDTOs);
}
```

Como vemos, se han declarado 4 métodos: toDTO(), toDTOList() cuya intención es transformar un objeto Usuario en un UsuarioDTO o una lista de Usuario en una lista de UsuarioDTO. Luego están los otros dos métodos para hacer la transformación inversa

Para que MapStruct actúe tenemos que anotar el interface con `@Mapper`

Observar que para los datos que tienen el mismo nombre y el mismo tipo NO tenemos que hacer nada, MapStruct se da cuenta y nos copiará la información entre esos atributos. Aquellos que tienen una diferencia tenemos que ayudarle con la anotación `@Mapping`

Vamos a entender la linea de mapping:

```
@Mapping(target = "fechaCreacion", expression = "java(usuario.getFechaCreacion() != null ? new
java.util.Date(usuario.getFechaCreacion()) : null)")
```

La etiqueta: target le indica cuál es el atributo que queremos establecer. En este caso es: UsuarioDTO.fechaCreacion. La etiqueta expression permite definir una sentencia java (observar que de hecho le especificamos que es una sentencia java con: `"java( … )"`) que hará el trabajo. En este caso le estamos diciendo que si el atributo no es nulo y contiene: numero, creamos un nuevo objeto: new Date(numero) que es la forma para obtener un Date de un unix epoch

Anexo: Converters

Hay formas para automatizar la conversión de atributos de la base de datos en otro tipo de atributos en las entity (ejemplo número unix time epoch en DDBB y objeto Date en la entity java) Que se hacen innecesarios si estamos separando el modelo/domain de las clases que hacen la persistencia (Entity) simplemente usaríamos mapper para pasar de uno a otro. Pero para casos en los que no estemos haciendo la separación (proyectos sencillos) tenemos la opción del

Interfaz: AttributeConverter<>

Así la siguiente clase Nos convierte de un número (Long) a un objeto Date:

```
@Converter(autoApply = true)
class DateToLongConverter implements AttributeConverter<Date, Long> {

    @Override
    public Long convertToDatabaseColumn(Date date) {
        return (date == null) ? null : date.getTime();
    }

    @Override
    public Date convertToEntityAttribute(Long timestamp) {
        return (timestamp == null) ? null : new Date(timestamp);
    }
}
```

Ahora en nuestra Entity informamos de que queremos aplicar el conversor en el atributo que trabaja como una fecha:

```
      @Convert(converter = DateToLongConverter.class)
      private Date fechanacimiento;
```