

Implementa la seguridad de tu API Rest con Spring Boot

BIENVENIDOS!

Soy Luis Miguel López Magaña

Me dedico a dar clase en Ciclos Formativos de Grado Superior



www.linkedin.com/in/luismi-lopez



[@LuisMLopezMag](https://twitter.com/LuisMLopezMag)

Requisitos para realizar el curso

- Se basa en el curso *Desarrollo de un API REST con Spring Boot y Elementos Avanzados en tu API REST con Spring Boot*

Recomendable...

- Curso de Spring Core
- Curso de Spring Boot y Spring Web MVC
- Conocimientos de Java
- Recomendable tener conocimientos de JPA/Spring Data Jpa (Curso de JPA e Hibernate y Curso de Spring Framework).

¿Qué vamos a aprender?

- Los elementos necesarios de Spring Security para tu API REST.
- En qué consiste la autenticación y la autorización.
- Diferentes posibilidades de implementar la seguridad.
- Implementar la seguridad básica
- Implementar la seguridad con Json Web Tokens
- Implementar la seguridad con OAuth 2.0

Contenidos

1. Introducción
2. Gestión de usuarios
3. Seguridad básica
4. Seguridad JWT
5. Seguridad con OAuth 2.0

Prácticas

- Iremos practicando la sintaxis en casi todas las lecciones.
- Completaremos un proyecto que completamos en el curso *Elementos avanzados en tu API REST con Spring Boot*.
- En algunas lecciones encontrarás un reto para poder poner en práctica tus conocimientos.

Cursos que puedes hacer al terminar

- Si aún no lo has hecho, el curso de Spring Boot y Spring MVC
- JPA e Hibernate
- Arquitecturas monolíticas y microservicios
- Curso de Istio
- Simplificando la seguridad de tu aplicación con Istio

Introducción a Spring Security

Implementa la seguridad de tu API Rest con Spring Boot

Spring Security

- Proyecto paraguas (integra muchos proyectos)
- Ofrece servicios de seguridad para aplicaciones Java EE
- Integración sencilla inmediata con proyectos Spring MVC a través de Spring Boot.
- Responde a dos cuestiones:
 - *Autenticación*: ¿quién eres?
 - *Autorización*: ¿para qué tienes permiso?

pom.xml

- Añadiendo dependencias starter

```
<dependency>
```

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

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

```
</dependency>
```

Spring Security

- Gran aceptación por la comunidad de desarrolladores por su *flexibilidad* en los modelos de autenticación.
- Rápida integración sin necesidad de una migración de los sistemas a un entorno de terceros.
- Plataforma abierta y en constante evolución.

Módulos

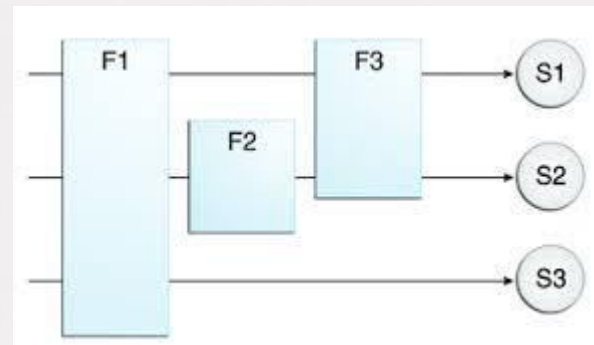
- Desde Spring Security 3, el código se encuentra dividido en diferentes .jars, para separar claramente funcionalidades y dependencias de terceros.
- Algunos de ellos son:
 - Core: contiene los elementos centrales de autenticación y control de acceso.
 - Web: contiene los filtros y el código que articula la infraestructura de seguridad de una aplicación web.

Módulos

- Algunos de ellos son:
 - LDAP: provee los mecanismos necesarios para la autenticación vía LDAP.
 - OAuth 2.0 Core: provee las clases e interfaces para dar soporte al framework OAuth 2.0 y OpenID Connect Core 1.0
 - OAuth 2.0 Client: proporciona el soporte cliente para OAuth 2.0 y OpenID Connect Core 1.0.

Java Filter

- Funcionalidad que se coloca entre el cliente y un servlet.
- Permite dejar pasar una petición, rechazarla o añadir una determinada funcionalidad.
- Uno de sus usos clásicos es la seguridad.



Seguridad web: Filtro

- Durante todo el curso desarrollaremos aplicaciones web (REST) basadas en el API Servlet de Java.
- Spring Security se integra con el contenedor de servlets utilizando un filtro (Filter) estándar.
- Solo necesitamos un contenedor de servlets para utilizar Spring Security.
- De hecho, no es necesario utilizar Spring para poder usar Spring Security:O

Spring Boot y Spring Security

- Al ejecutar un proyecto Spring Boot con la dependencia de Spring Security, suceden varias cosas *automáticamente*.
 - Se habilita la configuración por defecto, a través de un filtro, llamado *springSecurityFilterChain*.
 - Se crea un bean de tipo *UserDetailsService* con un usuario llamado *user* y una contraseña aleatoria que se imprime por consola.
 - Se registra el filtro en el contenedor de servlets para todas las peticiones.

Spring Boot y Spring Security

- Aunque no ha configurado mucho, tiene muchas consecuencias
 - Requiere autenticación para interactuar con nuestra aplicación
 - Genera un formulario de login por defecto.
 - Genera un mecanismo de logout
 - Protege el almacenamiento de la contraseña con BCrypt.
 - Prevé contra ataques CSRF, Session Fixation, Clickjacking...
 - ...

Autenticación y Autorización

Implementa la seguridad de tu API Rest con Spring Boot

Autenticación y autorización

- La seguridad de una aplicación suele reducirse a dos problemas más o menos independientes
 - Autenticación: ¿quién es usted?
 - Autorización: ¿qué se le permite hacer?
- En ocasiones se llama control de acceso a la autorización.

Autenticación

Autenticación

- Spring Security proporciona un interfaz, *AuthenticationManager*, que implementa el patrón estrategia.

```
public interface AuthenticationManager {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
}
```

Autenticación

- Un *AuthenticationManager* puede hacer tres cosas con su único método:
 - Devolver un *Authentication* (normalmente con *authenticated=true*)
 - Lanzar una excepción de tipo *AuthenticationException*
 - Devolver *null*

Autenticación

- La implementación más usada de *AuthenticationManager* es *ProviderManager*, el cual delega en una cadena de instancias de tipo *AuthenticationProvider*.
- Un *AuthenticationProvider* se parece a un *AuthenticationManager*, ya que solo añade un nuevo método que permite verificar si la instancia soporta un determinado tipo de *Authentication*.
- Un *ProviderManager* puede soportar diferentes mecanismos de autenticación en una sola aplicación.

Autenticación

- Un *ProviderManager* puede tener un padre, que puede consultar si todos sus *provider* han devuelto null.
- Si no hay un padre disponible, una respuesta null se transforma en una excepción (*AuthenticationException*).
- En ocasiones, se puede tener grupos de recursos (por ejemplo, recursos web en un determinado path), y cada grupo tener su propio *AuthenticationManager*. Si establecemos una jerarquía, algunos grupos podrían compartir un padre como mecanismo global de autenticación.

AuthenticationManagerBuilder

- Spring Security ofrece algunos mecanismos rápidos de configuración de un *AuthenticationManager*.
- El más común es el uso de un *AuthenticationManagerBuilder*. Este permite configurar rápidamente
 - Autenticación en memoria
 - JDBC
 - LDAP
 - Un servicio de UserDetailsServices personalizado

Autorización o control de acceso

Autorización

- Una vez que la autenticación ha sido exitosa, pasamos al control de acceso, a través de la interfaz *AccessDecisionManager*.
- Hay 3 implementaciones de esta interfaz, y todas delegan en una cadena de *AccessDecisionVoter* (algo así como el *ProviderManager*).
- Un *AccessDecisionVoter* considera un *Authentication* y un *objeto* seguro (este objeto es genérico y puede representar cualquier cosa, como un recurso web). El objeto seguro es decorado a través de una colección de *ConfigAttributes*.

Autorización

- Los *ConfigAttributes* decoran un objeto con metadatos para determinar el nivel de permisos requeridos para acceder a él.
- La interfaz *ConfigAttribute* es muy sencilla, y tiene solo un método que devuelve un String.
- Esta cadena codifica, de alguna forma, la intención del propietario del recurso. Por ejemplo, es típico el nombre de un rol, como *ROLE_ADMIN* o *ROLE_AUDIT*.

Autorización

- También es común usar *ConfigAttributes* basados en expresiones SpEL, como *isAuthenticated()* o *hasRole('THEROLE')*.

Configuración de la autorización

- Se pueden configurar estos *AccessDecisionVoter* a través de diferentes mecanismos, que utilizaremos a lo largo del curso. Entre otros:
 - Extendiendo la clase *WebSecurityConfigurerAdapter* y el uso de *AntMatchers* (patrones de rutas)
 - A través de anotaciones (*@PreAuthorize* y *@PostAuthorize*)

Algunas clases e interfaces de Spring Security

Implementa la seguridad de tu API Rest con Spring Boot

Clases e interfaces

- En la lección anterior hemos nombrado algunas clases o interfaces.
- Dada la amplitud de Spring Security, no podemos conocer todas.
- Nos centramos en las más comunes o las que más usaremos.

WebSecurityConfigurerAdapter

- Clase base para nuestra clase de configuración de seguridad web.
- Suele venir acompañada de @Configuration + @EnableWebSecurity.
- Métodos convenientes para configurar la autenticación y la autorización.

@EnableWebSecurity

- Sirve para conmutar (apagar) la configuración por defecto aplicada por Spring Boot, y añadir la nuestra.
- Se utiliza anotando una clase que extienda a *WebSecurityConfigurerAdapter*.

Authentication

- Interfaz que extiende a `Principal`
- Representa un token para realizar la autenticación, o para un principal una vez autenticado.
- Normalmente se almacena en el contexto de seguridad (*SecurityContext*) manejado por el (*SecurityContextHolder*)
- Spring Security tiene decenas de clases que lo implementan
- **Interfaz nuclear en la autenticación.**

AuthenticationManagerBuilder

- *Builder* utilizado para construir un `AuthenticationManager`.
- Permite construir, fácilmente, un `AuthenticationManager` en memoria, LDAP, JDBC o con `UserDetailsService`.
- Se suele configurar sobrescribiendo el método *`configure(AuthenticationManagerBuilder)`* de la clase *`WebSecurityConfigurerAdapter`*.

UserDetails

- Interfaz que representa la información nuclear de un usuario.
- Almacena la información que posteriormente será encapsulada en un objeto de tipo *Authentication*.
- Las implementaciones de esta interfaz deben verificar bien cada método, para saber qué atributos no deben ser nulos.
- Es implementado por la clase *org.springframework.security.core.userdetails.User*.

User

- Objeto modelo que incluye la información de un usuario obtenido por un UserDetailsService.
- Se puede usar directamente, extenderla o implementar la interfaz *UserDetails*.
- La implementación de equals y hashCode se basa en el atributo *username*.
- Incluye las *Authorities* del usuario.

GrantedAuthority

- Representa un privilegio individual.
- Se pueden usar con perspectiva de *grano fino*.
 - CAN_READ_SOME_ENTITY_PROPERTY
- El nombre es arbitrario.
- Solo un método, que devuelve la representación como String.
- En ocasiones, pueden representar a un rol con el prefijo *ROLE_*

SimpleGrantedAuthority

- Implementación concreta y muy básica de *GrantedAuthority*.
- Almacena una representación en un String de una *authority* concedida a un objeto de tipo *Authentication*.

UserDetailsService

- Interfaz que es capaz de cargar la información de un usuario.
- Se puede utilizar como DAO (Data Access Object).
- Es utilizado por DaoAuthenticationProvider (un *AuthenticationProvider* que obtiene la información de los usuarios a través de un *UserDetailsService*).
- Un solo método, *UserDetails loadUserByUsername(String)*.
- Se suele utilizar cuando almacenamos la información de los usuarios a través de Spring Data y el uso de entidades.

Posibilidades para implementar la seguridad en un API Rest

Implementa la seguridad de tu API Rest con Spring Boot

Mecanismos de autenticación en una web con UI

- La mayoría de las webs que utilizamos suelen proveer un mecanismo para la autorización a través de un formulario de login.
- Normalmente se les proporcionan dos datos: usuario y contraseña.
- El servidor suele ser el encargado de almacenar la sesión (usuario activo en la aplicación).
- ¿Qué hacer si nuestra aplicación no tiene UI? Como por ejemplo, con un API REST.

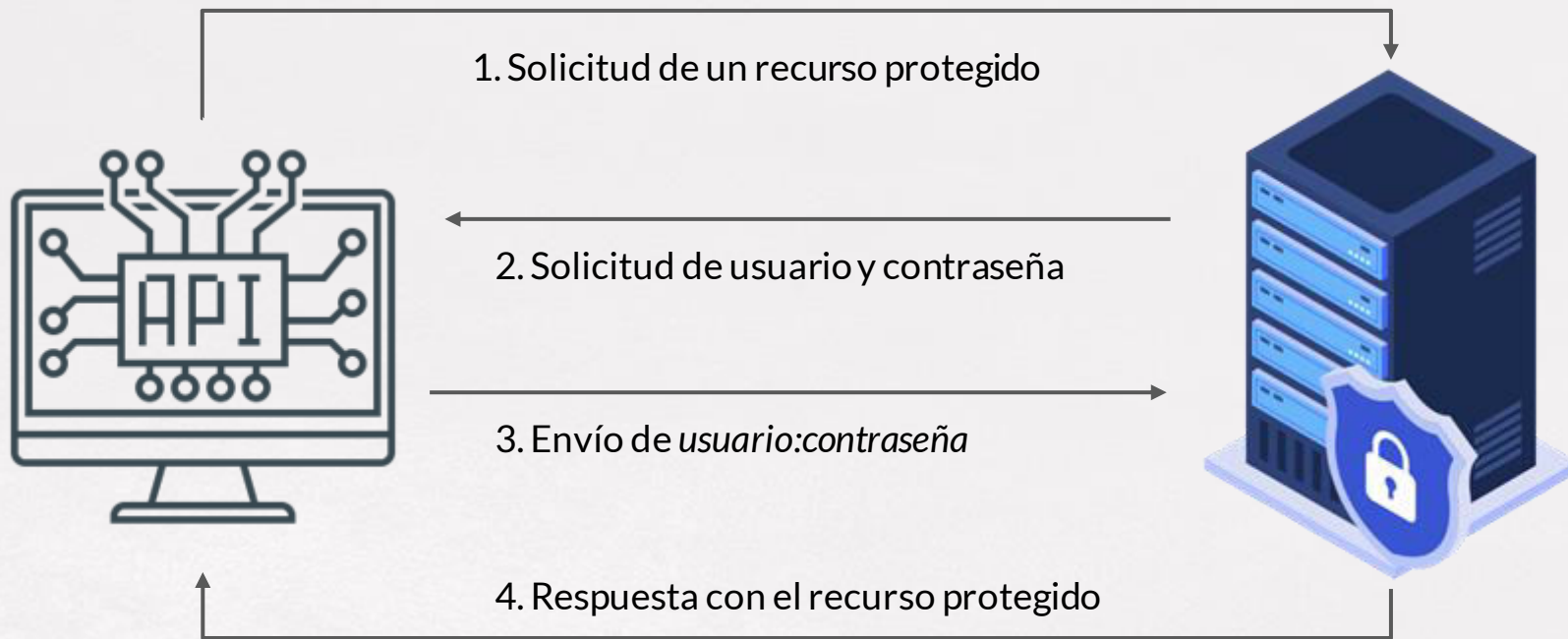
Diferentes mecanismos de autenticación en un API REST

- Básica (*basic*)
- JWT (Json Web Tokens)
- OAuth 2.0

Autenticación básica

- Mecanismo más elemental de autenticación a través de HTTP.
- Definido en el RFC 1945 y RFC 2617
- *No es elegante, pero cumple su función.*
- Es simple, pero poco fiable.
- No obliga al uso de cookies ni de formularios de acceso.

Autenticación básica



Autenticación utilizando JWT

- Json Web Token.
- Realmente no es un estándar de autenticación.
- Se trata de un estándar para la creación de tokens de acceso que permiten propagar la identidad y privilegios.
- La información puede ser verificada y confiable, porque está firmada digitalmente.
- No obliga a que el servidor maneje la sesión.

Autenticación utilizando JWT



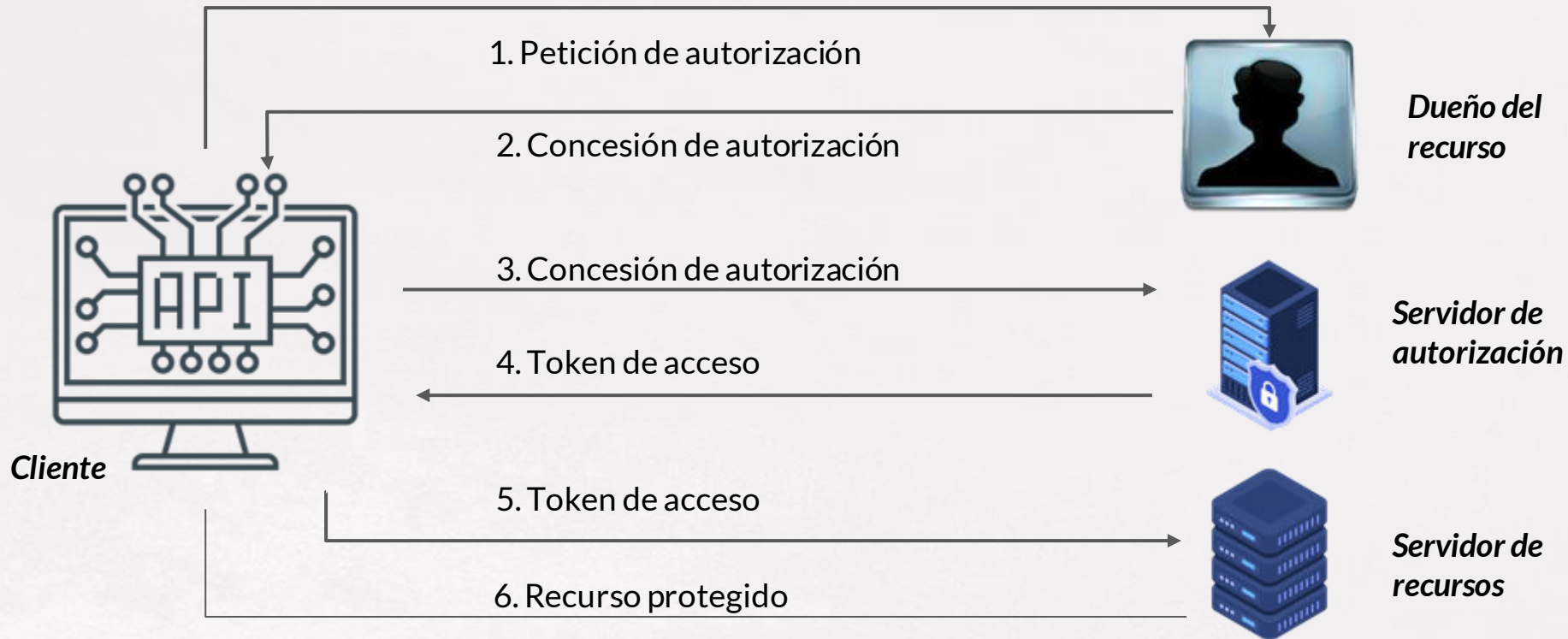
Autenticación OAuth 2.0

- Estándar abierto para la autorización de APIs.
- Permite compartir información entre sitios sin compartir de la identidad.
- Mecanismo utilizado por grandes compañías, como Google, Facebook, Microsoft, Twitter y Github.
- Implementa diferentes flujos de autenticación: *authorization code flow*, *resource owner password credential flow*, *implicit flow*, ...

Autenticación OAuth 2.0

- Se definen varios roles
 - Dueño del recurso
 - Cliente
 - Servidor de recursos protegidos
 - Servidor de autorización.

Autenticación OAuth 2.0



Modelo de usuario y rol

Implementa la seguridad de tu API Rest con Spring Boot

Modelo de usuario

- Representará a una persona que utilice nuestro sistema.
- Información básica: nombre de usuario, contraseña y avatar.
- Además, el rol o roles que tiene dicho usuario.
- Lo implementamos como una entidad de JPA para almacenarlo fácilmente en una base de datos.

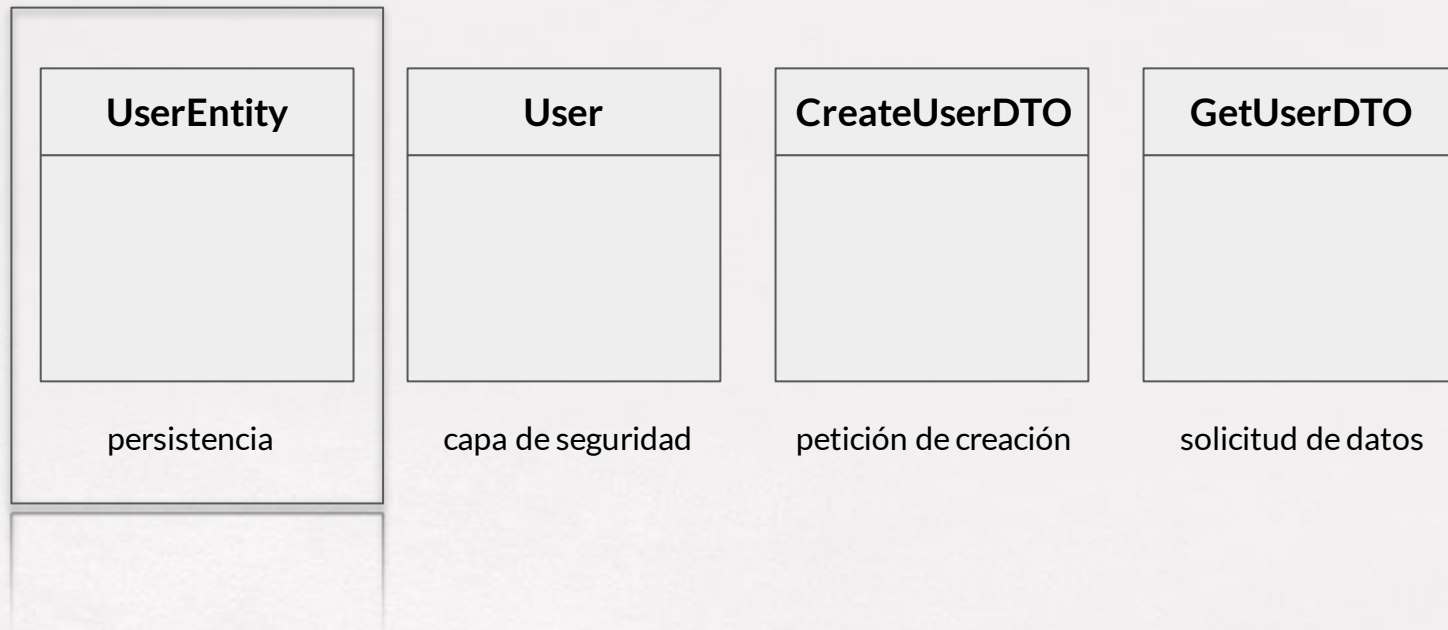
Alternativas

- Si nuestro modelo de usuario implementa *UserDetails*
 - **Desventaja:** más acoplado a Spring Security (y sus posibles cambios).
 - **Ventaja:** más integrado con Spring Security (Authentication = nuestro modelo de usuario).

Alternativas

- Si nuestro modelo de usuario no implementa *UserDetails*
 - **Desventaja:** más integrado con Spring Security (necesitamos transformar, en algún punto, nuestra entidad usuario en algo que implemente a *UserDetails*).
 - **Ventaja:** menos acoplado con Spring Security (nos afectarán menos los cambios).
- Escogemos la primera alternativa.

Múltiples clases para gestionar los usuarios



¡A por el código!

- Clase entidad para *UserEntity*
- Enumeración *UserRole*
- Anotación en una clase de configuración (por ahora nos vale la clase principal).

Reto

- Plantea algunos campos más que pudiera tener la clase
- Revisa los campos y añade las anotaciones de validación que consideres oportunas (curso de Spring Boot)
- Añade los campos necesarios para gestionar de verdad los atributos *accountNonExpired*, *accountNonLocked*, *credentialNonExpired* o *enabled*. Puedes revisar la documentación de la interfaz `UserDetails`.

Repositorio y servicios

Implementa la seguridad de tu API Rest con Spring Boot

Repositorio

- Creamos un repositorio que hereda de *JpaRepository*
- Añadimos una consulta (derivada del nombre) para obtener un usuario por su nombre de usuario.

Servicio Base

- Se propone una clase base para cualquier servicio de la aplicación.
- Sirve como envoltorio del repositorio (y así, en los controladores no utilizamos directamente los repositorios, usando solamente servicios).

Servicio

- Añadimos el envoltorio para el método de consulta creado en el repositorio.
- Posiblemente, en el futuro *refactoricemos* para añadir más métodos.

Servicio *UserDetailsService*

Implementa la seguridad de tu API Rest con Spring Boot

UserDetailsService

- Interfaz que es capaz de cargar la información de un usuario.
- Se puede utilizar como DAO (Data Access Object).
- Es utilizado por DaoAuthenticationProvider (un *AuthenticationProvider* que obtiene la información de los usuarios a través de un *UserDetailsService*).
- Un solo método, *UserDetails loadUserByUsername(String)*.
- Se suele utilizar cuando almacenamos la información de los usuarios a través de Spring Data y el uso de entidades.

UserDetailsService

- A menudo hay cierta confusión al respecto *UserDetailsService*.
- **Es puramente un DAO para datos de usuario** y *no realiza otra función* que no sea suministrar esos datos a otros componentes dentro del framework.
- En particular, **no autentica al usuario**, lo que sí hace *AuthenticationManager*.
- En muchos casos, tiene más sentido implementar *AuthenticationProvider* directamente si se necesita un proceso de autenticación personalizado.

UserDetailsService

- Como nuestra clase modelo *UserEntity* implementa la interfaz *UserDetails*, el cuerpo de este método es muy sencillo.
- Si no se encuentra el usuario, lanzamos una excepción de tipo *UsernameNotFoundException*.

Controlador de registro

Implementa la seguridad de tu API Rest con Spring Boot

Controlador

- El controlador para la gestión de usuarios deberá permitir crear nuevos usuarios.
- Estos usuarios se crearán por defecto con el rol más básico.
- Su contraseña se debe guardar en la base de datos, pero cifrada.

Refactorización en el servicio

- Necesitamos un método que almacene un nuevo usuario
 - Deberá asignar el rol (*UserRole.USER*)
 - Deberá encriptar la contraseña
- Para encriptar la contraseña, necesitamos un bean *encriptador*.
 - Usamos el algoritmo BCrypt.

Controlador

- Recibimos los nuevos datos a través de una instancia de *UserEntity*.
- Almacenamos el usuario y devolvemos la instancia almacenada.

Refactorización para utilizar DTO

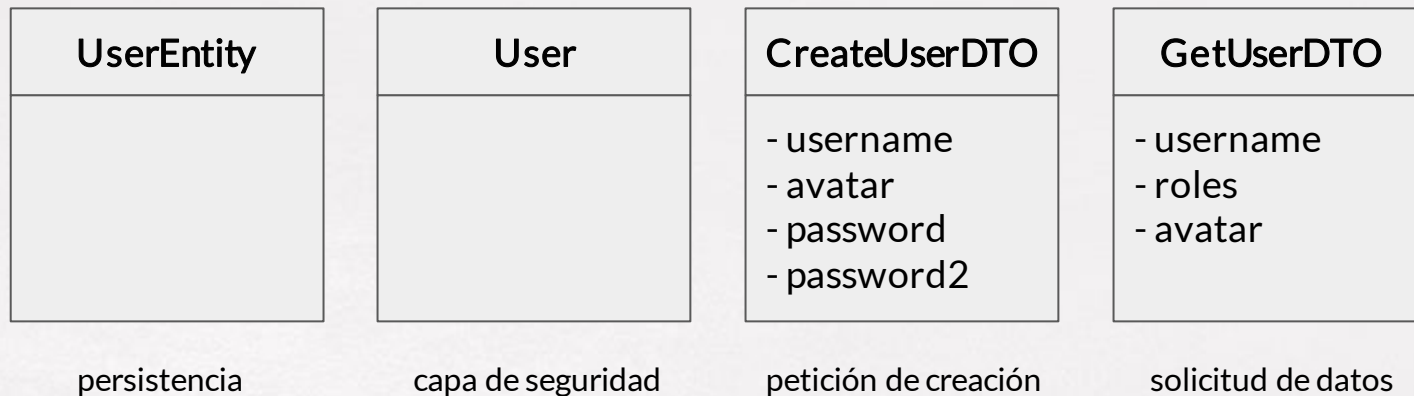
Implementa la seguridad de tu API Rest con Spring Boot

Clases modelo

- Nuestro modelo de usuario no es conveniente para crear un nuevo usuario.
 - Sería bueno recibir la contraseña por duplicado, para que podamos verificar si hay algún error o no.
 - Solo necesitamos algunos datos, no todos
- Tampoco es un buen candidato para la salida
 - Algunos datos aparecen más de una vez, dada la estructura del modelo (roles, authorities, ...)

Patrón DTO

- *Data Transfer Object*
- Sirve para transportar datos entre capas de un sistema



Gestión de errores

- Gestionamos en servidor la validación de la contraseña del usuario (que ambas contraseñas sean iguales)
- También gestionamos el intento de insertar un usuario duplicado (*@Column(unique = true)*)
- Tratamiento global y tratamiento local

Clases en la gestión de errores

- *NewUserWithDifferentPasswordsException*
 - Excepción para manejar la validación de la contraseña.
- *ApiError, ApiErrorAttributes*
 - Modelo de error para enviar una respuesta al cliente
- *GlobalControllerAdvice*
 - Tratamiento global de errores

Cambios en el código

- Nuevas clases CreateUserDto y GetUserDto
- Conversor de UserEntity a GetUserDto
- Cambios en el servicio y el controlador

Reto

- Es un buen momento para poder integrar el proyecto base (*00_ProyectoBase*) con el trabajo realizado hasta ahora.
- No servirá como nueva base para aprender los diferentes mecanismos de autenticación.
- El punto de unión entre ambos proyectos es modificar el modelo de pedido, para que el *cliente* sea un *UserEntity*.

Seguridad básica: ¿en qué consiste?

Implementa la seguridad de tu API Rest con Spring Boot

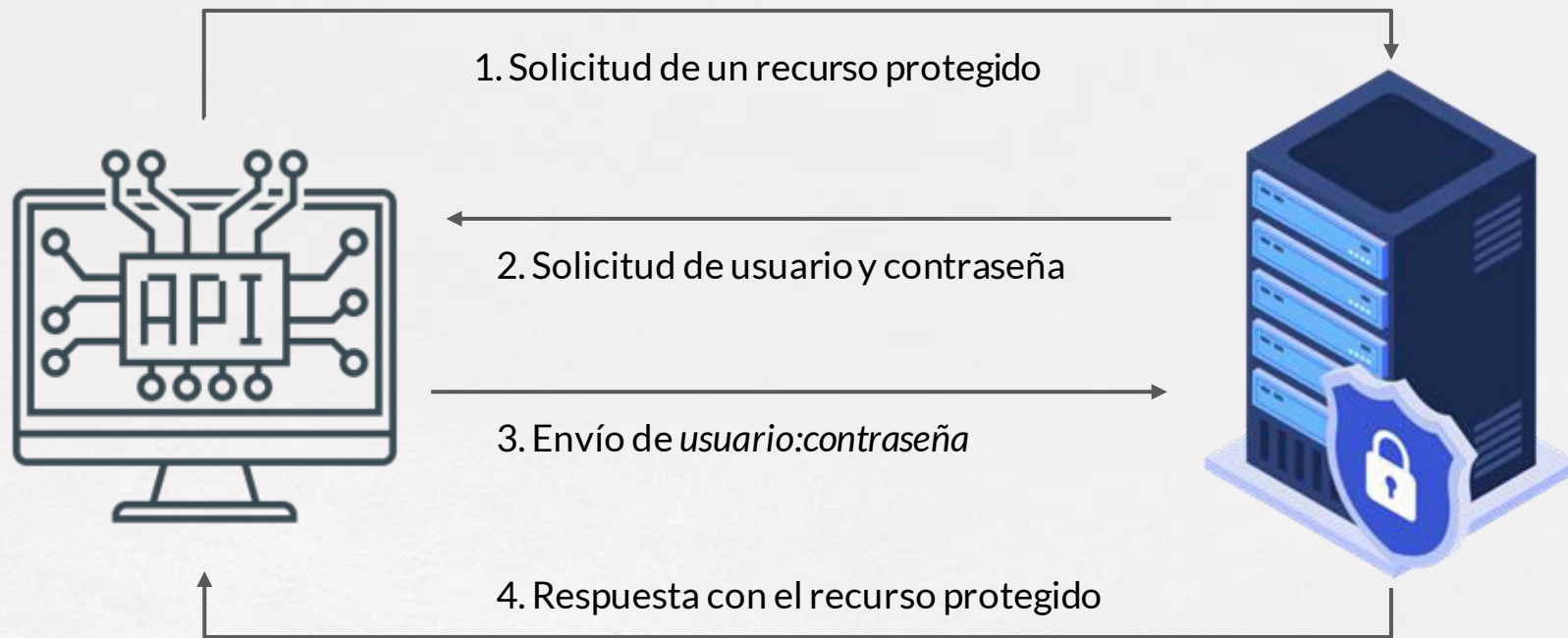
Autenticación básica

- Método para que un cliente (o navegador web) pueda enviar las credenciales de un usuario (usuario y contraseña) al servidor.
- Definida en la especificación de HTTP (RFC 1945, RFC 2617)
- Simple de implementar, pero puede ser no adecuada en muchas situaciones.

Autenticación básica

- No está pensado para canales públicos
- Las credenciales se envían en Base64 (no es un cifrado, solo una codificación).
- Si trabajamos con HTTP, “cualquiera” las podría descifrar.
- No obliga al uso de cookies ni de formularios de acceso.

Autenticación básica



Autenticación: lado cliente

- Se utiliza el encabezado *Authorization*
- La cabecera se construye siguiendo estos pasos
 - Se concatenan nombredeusuario, :, y contraseña
 - La cadena se codifica en Base64
 - el método de autorización es Basic, seguido de un espacio.
- Un ejemplo sería `Authorization: Basic dXNlcjoxMjM0` siendo las credenciales *user* y *1234*.

Autenticación: respuesta del servidor

- Si la autenticación tiene éxito, se devuelve el recurso solicitado
- Si no, se debe devolver un código 401 No autorizado
- La respuesta incluirá además
 - Una cabecera WWW-Authenticate

`WWW-Authenticate: Basic realm="TheRealm"`

AuthenticationEntryPoint

- Se invoca cuando la autenticación falla
- Implementación por defecto: *BasicAuthenticationEntryPoint*
- Podemos (y lo haremos) proporcionar nuestra propia implementación.
- Además del código de respuesta (401) y la cabecera indicada por el RFC correspondiente, enviaremos un mensaje de error JSON.

Seguridad básica: implementación

Implementa la seguridad de tu API Rest con Spring Boot

Configuración de seguridad

- Debemos configurar nuestro mecanismo de autenticación (UserDetailsService)
- También debemos configurar el control de acceso.
- *Refactorizamos* el código del *Password Encoder* para sacarlo fuera, a otro bean (evitar referencia circular *SecurityConfig* → *UserDetailsService* → *UserEntityService* → *SecurityConfig* ...)

Control de acceso (autorización)

- ¿Quién puede hacer qué?
- Lo implementamos sobrescribiendo el método `WebSecurityConfigurerAdapter.configure(HttpSecurity http)`.
- Identificamos rutas, métodos HTTP y roles que pueden acceder.
- Indicamos que la autenticación será básica.
- Establecemos un `AuthenticationEntryPoint` personalizado.

AuthenticationEntryPoint personalizado

- Se va a encargar de responder cuando el usuario no haya conseguido autenticarse correctamente.
- Transformaremos a JSON la respuesta para completarla.

Y ahora, ¡al código!

Seguridad básica: refactorización

Implementa la seguridad de tu API Rest con Spring Boot

Algunos cambios

- Para poder integrar la seguridad con el funcionamiento de la aplicación necesitamos hacer algunos cambios, y añadir alguna funcionalidad.
- Usualmente, suele implementarse un *endpoint* para obtener los datos de usuario (perfil). Podría estar en */user/me*.
 - Debe devolver los datos del usuario que esté actualmente autenticado.

Algunos cambios en Pedidos

- Obtener todos
 - Un usuario con rol ADMIN puede obtener todos los pedidos
 - Un usuario con rol USER puede obtener **todos sus pedidos**.
- Nuevo pedido
 - Debemos asociar, como cliente que realiza el pedido, el usuario que actualmente está **autenticado**.

Reto

- Si no lo has hecho el resto de cursos sobre API REST con Spring Boot, puedes completar este proyecto
- *PedidoController*
 - Modificar un pedido (PUT)
 - Añadir/Eliminar una línea de pedido
 - Modificar la cantidad de una línea de pedido
 - Eliminar un pedido (DELETE)
 - Solo lo puede hacer el dueño o un ADMIN

Reto

- Todos los métodos de controlador en
 - *LoteController*
 - Modificar un lote (PUT)
 - Eliminar un lote (DELETE)

Reto 2

- Añade un nuevo rol, llamado CONTENT_MANAGER
 - Inserta un usuario de ejemplo con dicho rol
 - Permite que pueda realizar GET, POST y PUT sobre los productos y POST y PUT sobre lote.

Seguridad básica: despliegue y prueba

Implementa la seguridad de tu API Rest con Spring Boot

Comprobemos que nuestra API funciona

- POSTMAN o curl
- Probamos a hacer una petición sin autenticación
 - Debemos obtener un error 401
- Probamos a hacer una petición autenticados como USER
- Probamos a hacer una petición autenticados como ADMIN

POSTMAN

- Petición con datos correctos

The screenshot shows the Postman application interface. At the top, the request method is set to 'GET' and the URL is 'http://localhost:8080/user/me'. Below this, a tabbed interface shows 'Params', 'Authorization' (selected), 'Headers (1)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. In the 'Authorization' tab, the 'TYPE' is set to 'Basic Auth'. A warning message states: 'Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend you use variables.' The 'Username' field contains 'marialopez' and the 'Password' field contains 'Marialopez1'. The 'Show Password' checkbox is checked. A 'Preview Request' button is visible on the left side of the 'Authorization' tab.

GET http://localhost:8080/user/me

Params Authorization Headers (1) Body Pre-request Script Tests Settings

TYPE

Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Preview Request

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend you use variables

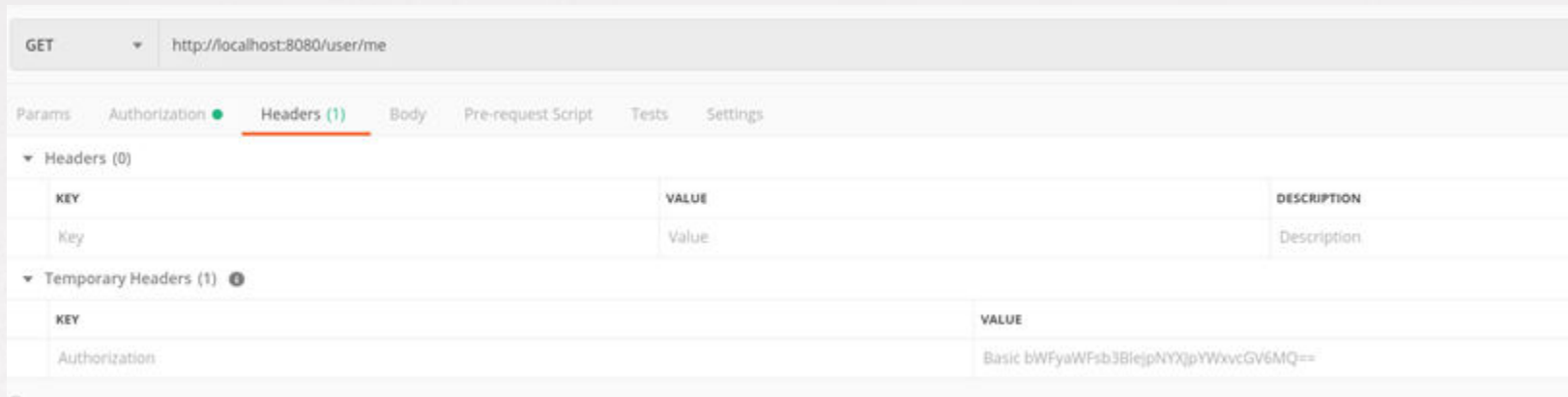
Username: marialopez

Password: Marialopez1

☒ Show Password

POSTMAN

- Si pulsamos *Preview Request* podemos ver la cabecera generada antes de enviar la petición.



GET http://localhost:8080/user/me

Params Authorization Headers (1) Body Pre-request Script Tests Settings

▼ Headers (0)

| KEY | VALUE | DESCRIPTION |
|-----|-------|-------------|
| Key | Value | Description |

▼ Temporary Headers (1) ⓘ

| KEY | VALUE |
|---------------|--------------------------------------|
| Authorization | Basic bWFyaWFsb3BlejpNYXpYWxcGV6MQ== |

POSTMAN *issue*

- A día de hoy, POSTMAN tiene algunas dificultades con la autenticación Básica
- En ocasiones, una vez ejecutado el programa, solo nos dejará utilizar las primeras credenciales que usemos.
- Las peticiones siguientes, aunque modifiquemos las credenciales, se realizarán con las primeras que hayamos utilizado.
- Mientras solucionan, podemos utilizar *curl*

Error 403

- Se produce cuando estamos correctamente autenticados, pero no tenemos privilegio (ROL) para hacer una determinada tarea.
- El error producido tiene el formato estándar
- Si lo queremos modificar, necesitamos proporcionar un *AccessDeniedHandler* personalizado.
- Refactorizamos para incluirlo, siguiendo el esquema utilizado con el *AuthenticationEntryPoint*.

JWT: En qué consiste

Implementa la seguridad de tu API Rest con Spring Boot

JWT

- JSON Web Token (RFC 7519)
- Es un mecanismo para propagar de forma segura la identidad (y *claims* o privilegios) entre dos partes.
- Los privilegios se codifican como objetos JSON.
- Estos objetos se usan en el cuerpo (*payload*) de un mensaje firmado digitalmente.

Token JWT

- Se trata de una cadena de texto con 3 partes *codificadas* en Base64
- Las partes están separadas por un punto
 - eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.ikFGEvw-Du0f30vBaA742D_wqPA5BBHXgUY6wwqab1w
- Las partes están separadas por un punto
- Podemos utilizar un *debugger* online para decodificarlo
 - <https://jwt.io/#debugger-io>

Token JWT

- Pero... ¿si la información se ve! *¿Esto es seguro?*

The diagram illustrates the encoding process of a JWT token. It is divided into two main sections: **Encoded** and **Decoded**.

Encoded: Shows the final encoded JWT token as a single long string of characters, including letters, numbers, hyphens, and underscores.

Decoded: Shows the token's structure and content. It is divided into three parts:

- HEADER:** Contains the algorithm used for signing (e.g., "HS256") and the token type (e.g., "JWT").
- PAYLOAD:** Contains the data being transmitted, such as user information (e.g., "username": "johnny", "role": "Manager").
- SECRET:** The secret key used for signing the token, represented as a base64-encoded string.

Token JWT

- Hemos dicho que tiene 3 partes
 - *Header*: indica el algoritmo y el tipo de token (HS256 y JWT)
 - *Payload*: datos del usuario y privilegios
 - Como nosotros generamos el token, podemos incluir todos los datos que estimemos convenientes.
 - *Signature*: firma para verificar que el token es válido (aquí radica el *quid* de la cuestión).

Firma de un token JWT

- La firma se construye de tal forma que podemos verificar que el remitente es quien dice ser, y que el mensaje no ha cambiado por el camino.
- Se construye como el HMACSHA256 de
 - Codificación en base64 de *header*
 - Codificación en base 64 de *payload*
 - Un secreto (establecido por la aplicación)

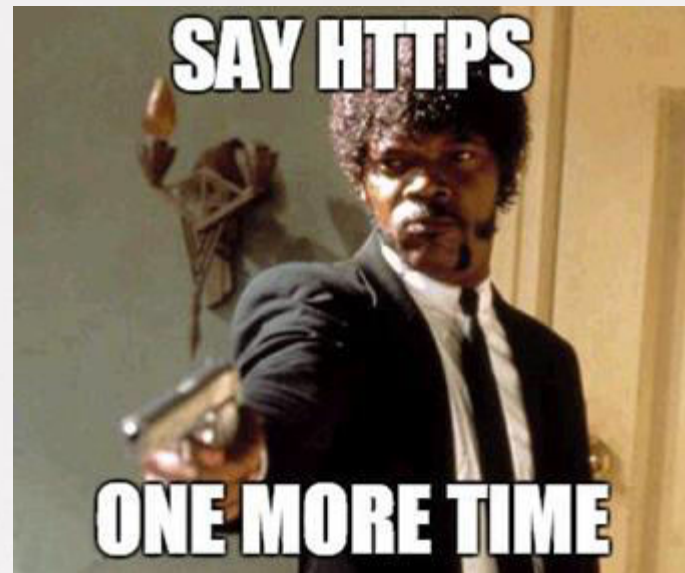


Firma de un token JWT

- Si alguien modifica el token por el camino:
...iOiJKV1QiLCJhbGci... a ...iOiRFH1QiLCJhbGci...
- La comprobación de la firma no será correcta
- No podemos confiar en el token recibido, y deberíamos denegarlo.
- Siempre debemos verificar la firma de un token recibido.

Token JWT seguro

- Con todo, el *header* y *payload* no están cifrados, solo codificados en base64.
- Esto nos invita a pensar que toda comunicación que hagamos debería ser con HTTPS para encriptar el tráfico.
- *En el fondo, siempre deberíamos utilizar HTTPS y un servidor con certificado.*



Ciclo de vida de un token JWT



JWT: Librerías necesarias

Implementa la seguridad de tu API Rest con Spring Boot

Spring Security y JWT

- Spring Security, de forma *nativa*, nos permite utilizar JWT en el contexto del uso del framework OAuth2.0
- Si queremos implementar la autenticación basada en JWT con el mecanismo que hemos visto en la lección anterior, necesitamos alguna librería externa.

JJWT: Java JWT

- Java / Android
- Integrable vía Maven/Gradle
- Uso ampliamente extendido
- Actualizada recientemente y con frecuencia

JJWT: funcionalidades

- Construir un token con sus diferentes partes:

String jws =

Jwts.builder().setSubject("Joe").signWith(key).compact();

- Construimos un JWT con un *claim sub* con el valor Joe
- Firmamos el JWT con un algoritmo adecuado (*HMAC-SHA256*)
- Lo compactamos en un *String*.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXLTJ5bqr60AID4

JJWT: funcionalidades

- Parsear un token, verificando así si está correctamente firmado
- Comprobar si
 - Ha expirado (*ExpiredJwtException*)
 - No está bien formado (*MalformedJwtException*)
 - Si la firma no es válida (*SignatureException*)

JWT: Implementación de la seguridad

Implementa la seguridad de tu API Rest con Spring Boot

Spring Security y JWT

- Como hemos visto en la lección anterior, Spring Security no nos proporciona todo lo necesario para trabajar con JWT.
- Implementaremos la seguridad a través de controladores y filtros:
 - *Controlador de autenticación*: recogerá el usuario y contraseña y, si es válido, construirá el token.
 - *Filtro de autorización*: recogerá un token y, si es válido, permitirá realizar la petición.

Más cambios

- *EntryPoint*
 - Modificaremos el EntryPoint con uno customizado (más adelante)
- Política de sesiones
 - Establecemos explícitamente que la política es sin sesiones.
- Seguridad a nivel de método
 - También la habilitaremos con `@EnableGlobalMethodSecurity`

JWT: AuthenticationEntryPoint

Implementa la seguridad de tu API Rest con Spring Boot

JWT: AuthenticationEntryPoint

- Se invoca cuando alguien realice un login incorrecto
- Debe devolver error *401 Unauthorized*
- Lo complementamos con un mensaje de error en JSON
- Implementamos la interfaz *AuthenticationEntryPoint* (en seguridad básica teníamos una clase base; aquí usamos directamente la interfaz).

JWT: Modelo de usuario y *UserDetailsService*

Implementa la seguridad de tu API Rest con Spring Boot

Modelo de usuario

- Nos sirve el que tenemos en el *proyecto base*.
- Más adelante, es posible que tengamos que refactorizar o crear algún nuevo Dto
 - Registro + login → *GetUserDto* con token
 - Login → *GetUserDto* con token o un nuevo *LoginUserDto*.

UserDetailsService

- Podemos comenzar con la implementación que tenemos en el *proyecto base*.
- Nuestro filtro necesitará un método para buscar un usuario por ID.
 - Podemos utilizar directamente *UserEntityService*
 - Creamos un envoltorio en ***UserDetailsService***
 - Si el día de mañana modificamos nuestro modelo de usuario, separándolo de UserDetails, solo tendremos que refactorizar este último método.

JWT: Manejo del token

Implementa la seguridad de tu API Rest con Spring Boot

JwtProvider

- Se encargará de
 - Generar un token a partir de un *Authentication* (un usuario logueado)
 - Obtener el ID de usuario a partir del payload de un token
 - Verificar si un token es válido.

Algunas clases a utilizar

- *JwtBuilder*
 - Nos permite construir un token JWT de una manera *fluida*.
 - Métodos
 - *setSubject*: indica el sujeto (para nosotros, el ID de usuario)
 - *setIssuedAt*: indica la fecha de creación del token
 - *setExpiration*: indica la fecha de expiración del token

Algunas clases a utilizar

- *JwtBuilder*
 - Más métodos
 - *claim*: permite indicar datos adicionales para el *payload*.
 - Añadiremos el *username* y los roles.
 - *setHeaderParam*: permite indicar parámetros para la cabecera del token.
 - *compact*: construye el token y lo serializa.

Algunas clases a utilizar

- *JwtBuilder*
 - Más métodos
 - *signWith*: permite firmar el token
 - A partir de la versión 0.10.X, es recomendable usar la firma *signWith(Key key, SignatureAlgorithm alg)*
- *Key*
 - *Keys.hmacShaKeyFor(byte[])* : permite generar un *SecretKey* basado en un array de bytes (listo para ser cifrado).

Y ahora, ¡al código!

JWT: Filtro de autorización

Implementa la seguridad de tu API Rest con Spring Boot

Filtro de autorización

- Encargado de revisar si una petición incluye un token JWT válido.
- Si verificamos que es válido, autenticamos al usuario en el contexto de seguridad.
- Algunas clases
 - *OncePerRequestFilter*: filtro que va a ejecutarse una vez en cada petición.
 - *UsernamePasswordAuthenticationToken*: una representación de *Authentication* muy simple, para presentar username y password.

Algoritmo del filtro

- Extraemos el token de la petición
- Si el token no es vacío y es válido
 - Obtenemos el ID de usuario del token
 - Obtenemos el usuario por su ID
 - Construimos un Authentication
 - Lo establecemos en el contexto de seguridad
- En otro caso, error, y la cadena de filtros no continua.

JWT: Modelo para el login y su respuesta

Implementa la seguridad de tu API Rest con Spring Boot

Petición de login

- Implementaremos más adelante el controlador.
- Este necesitará recibir las credenciales (*username* y *password*)
- Necesitamos un modelo para recibirlo en el método del controlador

Respuesta

- La respuesta puede ser tan solo el token obtenido o
- Podemos incluir algo más de información.
- Extenderemos la clase *GetUserDto* para incluir el token, y así enviar: nombre de usuario, avatar, nombre completo, roles y token.

Reto

- Prueba a jugar con la posible respuesta de la petición de login, para crear un modelo personalizado.
- También puedes pensar en crear una clase genérica, *JwtTokenResponse<InfoUsuario>*, donde *InfoUsuario* sea la información de usuario a enviar.
 - De esta forma, podemos enviar el token con diferente información acompañándole según nos interese en diferentes controladores (i.e.: login vs. signup)

JWT: Refactorización del controlador

Implementa la seguridad de tu API Rest con Spring Boot

AuthenticationController

- Petición de login
 - Logueamos al usuario vía *AuthenticationManager*
 - Almacenamos el *Authentication* en el contexto de seguridad
 - Devolvemos el usuario y token al cliente.
- Petición me
 - Igual de sencilla que en la autenticación básica

Reto

- Pedido
 - Basándonos en el código del ejemplo 12, en el que *refactorizamos* la funcionalidad de Pedido (controlador, servicio y repositorio), implementar dicha funcionalidad aquí, con JWT.
- Puedes implementarlo también para el resto de controladores.

Reto

- ¿No os dan rabia las aplicaciones en las que os tenéis que registrar e, inmediatamente, os piden las credenciales para *loguearos*?
- ¿No os gustan más aquellas en las que al registraros, os loguea automáticamente?
- Combinar el mecanismo de registro y login, para que el registro devuelva también el token.
- Es posible que merezca la pena que desaparezca *UserController* y pasar todo el código a *AuthenticationController*, pasando a llamarse este último *UserAuthenticationController*.

OAuth 2: ¿En qué consiste?

Implementa la seguridad de tu API Rest con Spring Boot

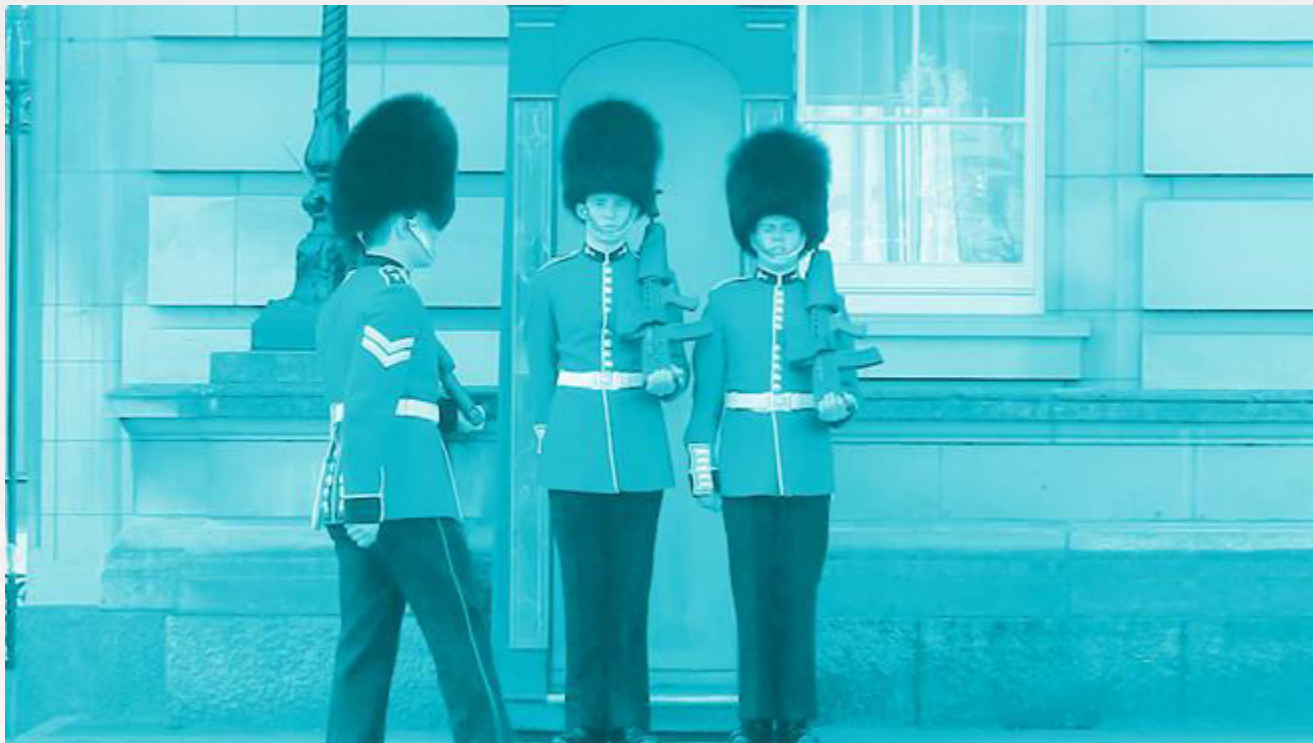
Autenticación OAuth 2.0

- Estándar abierto para la autorización de APIs.
- Permite compartir información entre sitios sin compartir de la identidad.
- Mecanismo utilizado por grandes compañías, como Google, Facebook, Microsoft, Twitter y Github.
- Implementa diferentes flujos de autenticación: *authorization code flow, resource owner password credential flow, implicit flow, ...*

¿Por qué surge OAuth?

- Paliar la necesidad del envío continuo de credenciales entre cliente y servidor.
- Integración con aplicaciones de terceros.
- El usuario delega la capacidad de realizar ciertas acciones en su nombre.
- Al desarrollar una aplicación, no tenemos necesidad de almacenar username/password del usuario.

Caso de uso



Caso de uso



Sign in

Sign in

☐ Remember me [Forgot Password?](#)

or

Login with your social media account

f Facebook **t Twitter** **G Google**

[Don't have an account? Sign up here!](#)

Algunos conceptos

- **OAuth 2.0** es un framework para la autorización (control de acceso) no para la autenticación.
- **Roles:** intervienen varios actores, que *desgranaremos* en las próximas lecciones.
- **Flujos:** en función de alguno de los tipos de actores, el flujo entre los diferentes actores podrá variar (i.e.: aplicaciones nativas vs. aplicaciones web).

OAuth2: Roles

Implementa la seguridad de tu API Rest con Spring Boot

Roles

- En OAuth2 Se definen varios roles
 - Dueño del recurso (*Owner*)
 - Cliente (*Client*)
 - Servidor de recursos protegidos (*Resource Server*)
 - Servidor de autorización (*Authorization Server*)

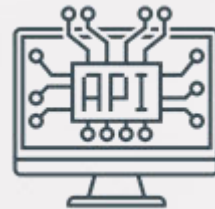
Dueño del recurso

- El propietario del recurso es el “usuario” que da la autorización a una aplicación, para acceder a su cuenta.
- El acceso de la aplicación a la cuenta del usuario se limita al “alcance” de la autorización otorgada (e.g. acceso de lectura o escritura).
- Se le llama el dueño de los recursos porque, si bien la API no es tuya los datos que maneja si lo son.



Cliente

- El cliente es la aplicación que desea acceder a la cuenta del usuario.
- Antes de que pueda hacerlo, debe ser autorizado por el usuario, y dicha autorización debe ser validada por la API.
- Este cliente puede ser una aplicación web, móvil, de escritorio, para Smart TV, un dispositivo IoT, **otra API**, etcétera.



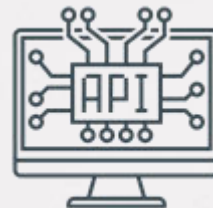
Servidor de autorización

- Es el responsable de gestionar las peticiones de autorización.
- Verifica la identidad de los usuarios y emite *tokens de acceso* a la aplicación cliente.
- En muchas ocasiones, estará implementado por un tercero conocido (*Facebook, Twitter, Github, Google, Okta,*)
- Puede formar parte de la misma aplicación que el servidor de recursos.



Servidor de recursos

- Será nuestra API, el servidor que aloja el recurso protegido al que queremos acceder.
- Puede formar parte de la misma aplicación que el servidor de autenticación.



Luces, cámara, acción

- Ahora, nos toca ver a estos 4 actores en acción.
- Lo hacemos en la siguiente lección.

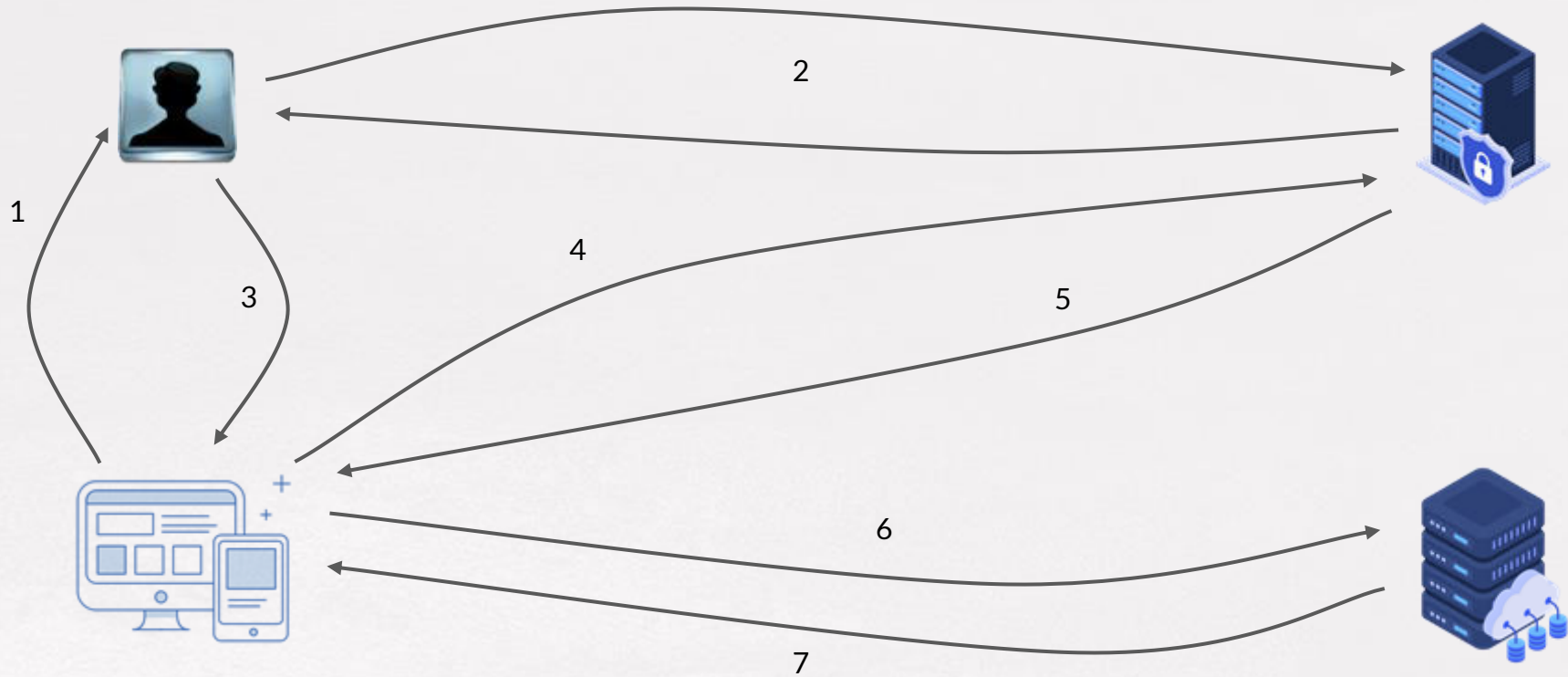
OAuth2: Flujo *abstracto* del protocolo

Implementa la seguridad de tu API Rest con Spring Boot

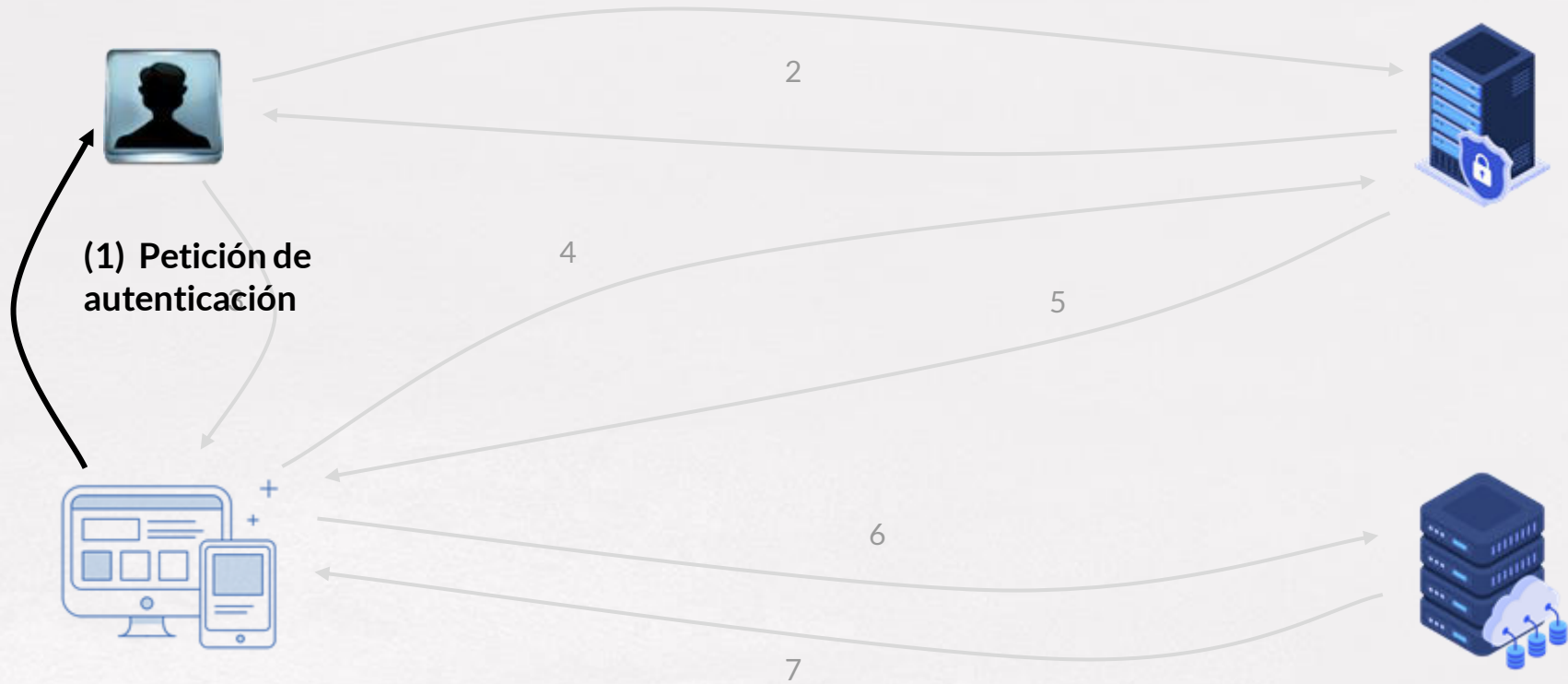
¿Por qué abstracto?

- No hay un único flujo en el protocolo
- En próximas lecciones veremos que diferentes tipos de actores pueden realizar diferentes concreciones del flujo que presentamos ahora.
- Vamos a conocer algunos de los elementos comunes del mismo.

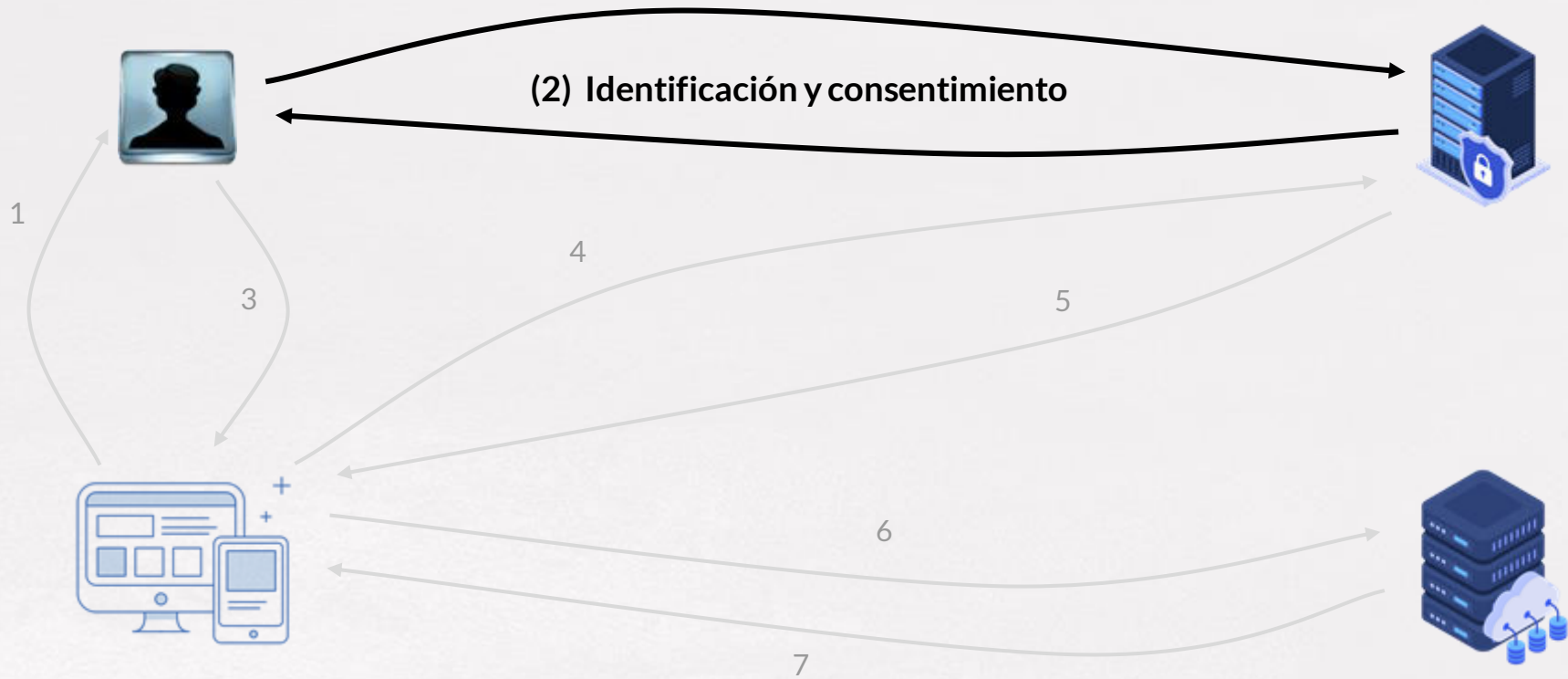
Flujo abstracto



Flujo abstracto



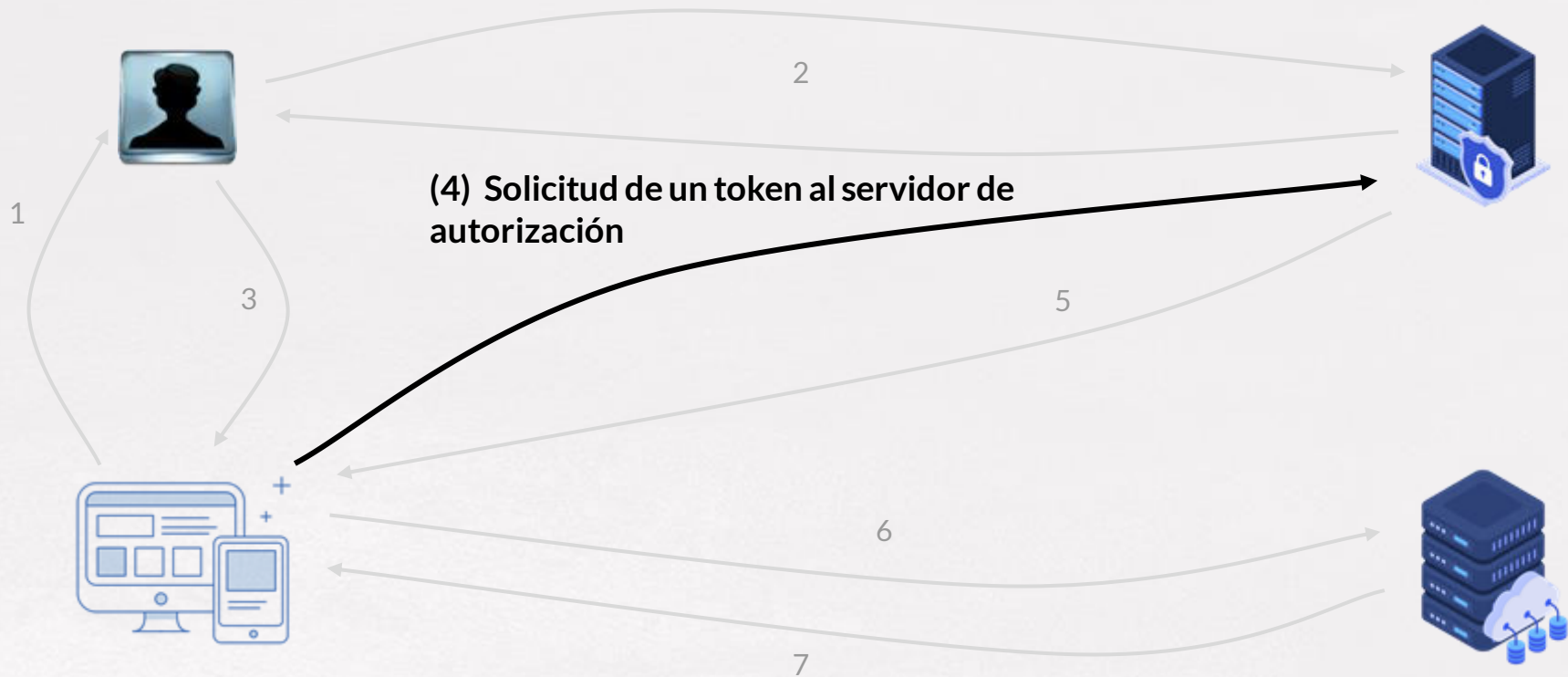
Flujo abstracto



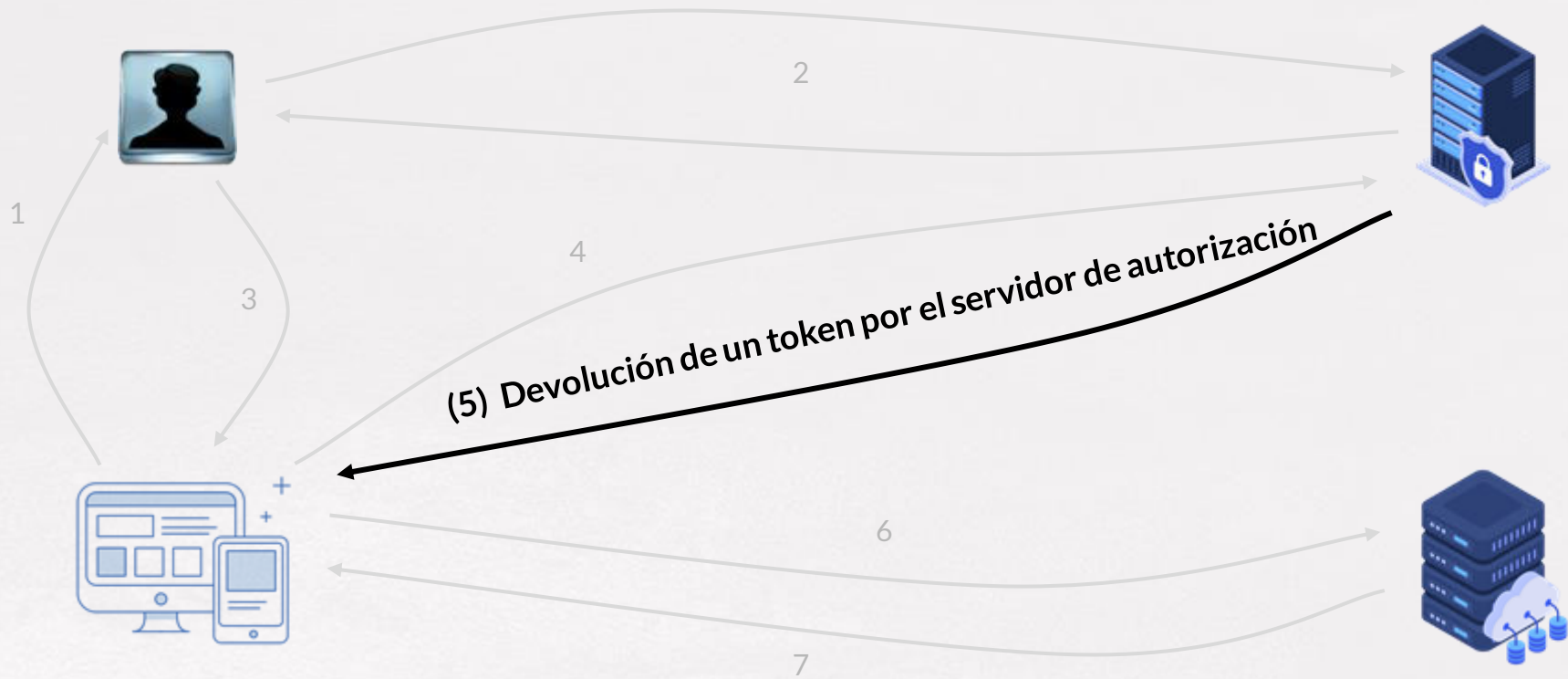
Flujo abstracto



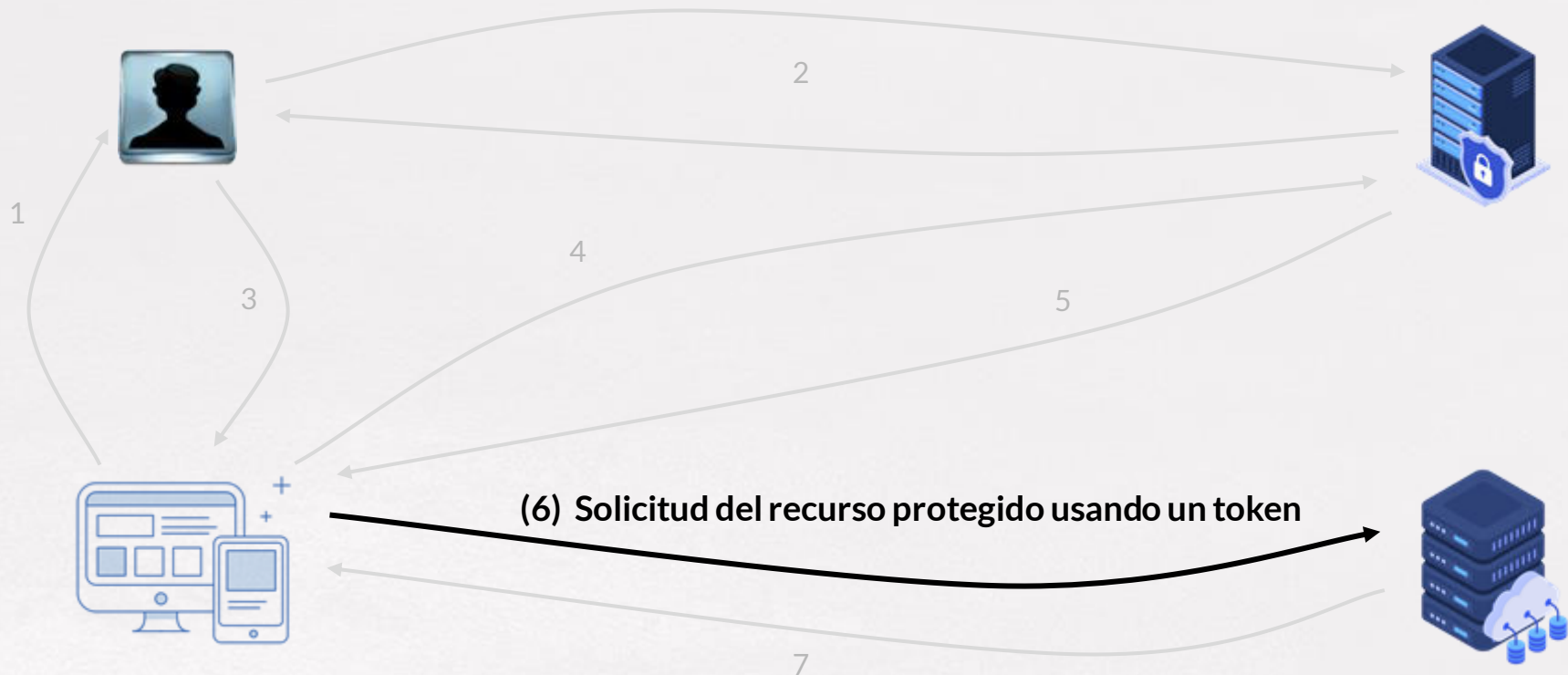
Flujo abstracto



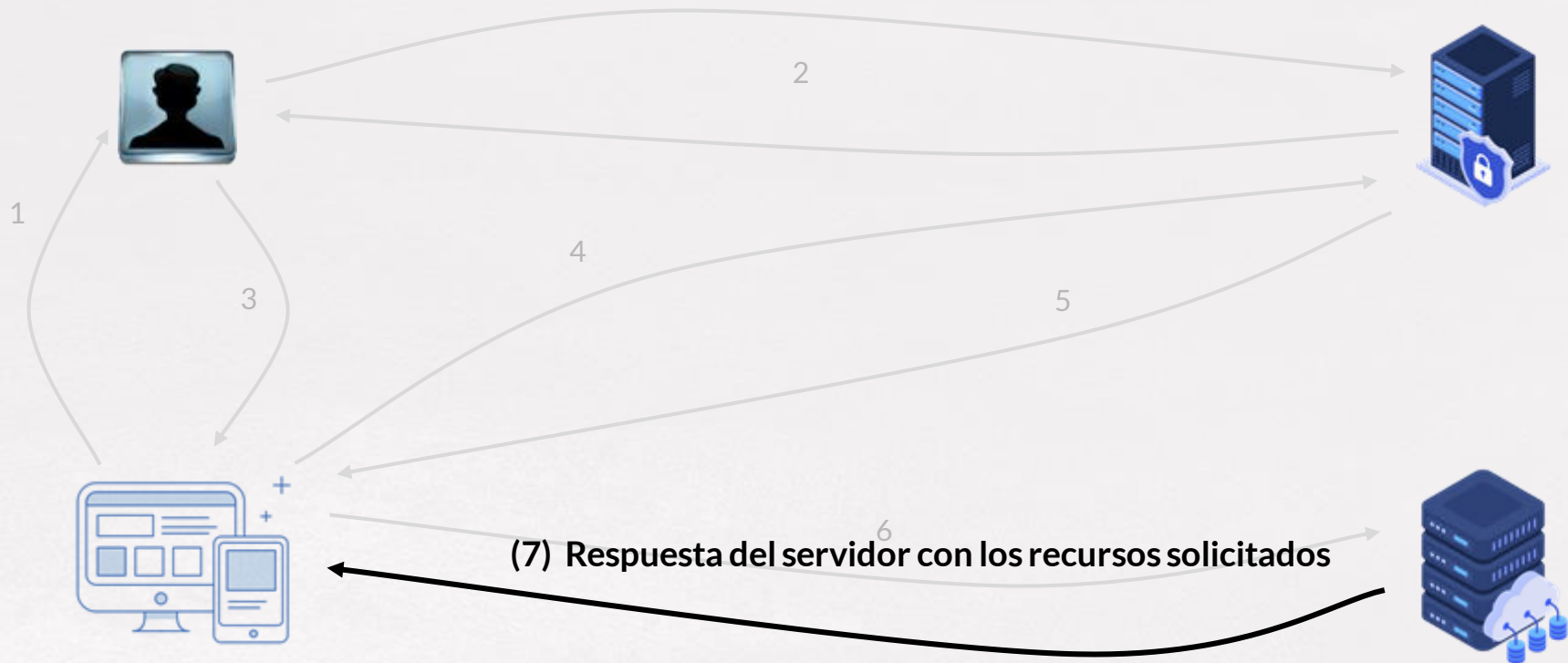
Flujo abstracto



Flujo abstracto

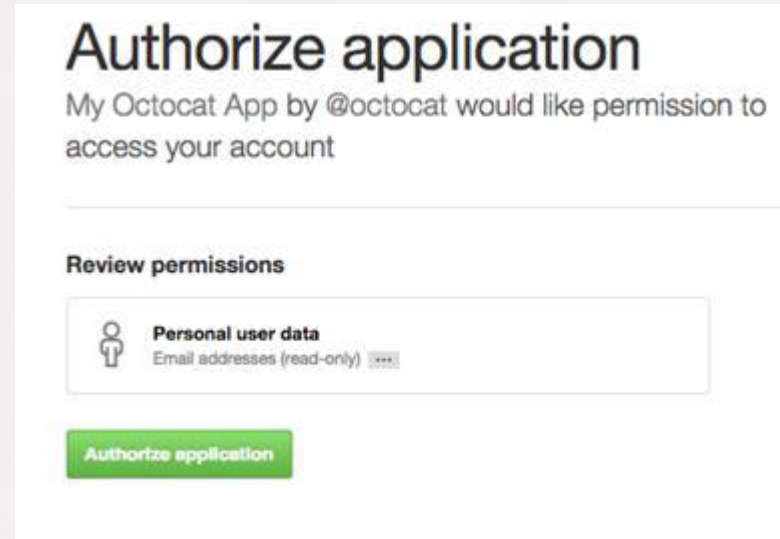


Flujo abstracto



Consentimiento

- Procedimiento que nos permite verificar qué pide la aplicación de nosotros (usuarios dueños de los recursos).
- OAuth 2.0 permite asegurarnos de que los usuarios sean conscientes, y de que el permiso se tenga que dar explícitamente.
- Seguro que es una página que has visto más de una vez.



Scopes o ámbitos

- Relacionado con el consentimiento
- Son los permisos que concedemos al cliente para realizar determinadas operaciones con un recurso protegido en nombre de un usuario.
- Deben ser lo más concretos posibles (para que no haya equívocos).

Endpoints

- Para autorizar la aplicación y obtener el token, necesitamos algunos servicios (endpoints) con los que interactuar.
 - **Authorization** (/oauth/authorize): para la autorización de la aplicación.
 - **Token** (/oauth/token): para la obtención del token.

OAuth2: Grant Types

Implementa la seguridad de tu API Rest con Spring Boot

Tipos de clientes

- **Clientes confidenciales:** son aquellos capaces de guardar una contraseña sin que esta sea accesible o expuesta (aplicaciones nativas, otra api, ...)
- **Clientes públicos:** son aquellos que no son capaces de guardar una contraseña y mantenerla a salvo (aplicaciones Javascript, Angular, ...)
- En función del tipo de cliente, necesitaremos implementar el flujo de OAuth2 de diferentes formas concretas.

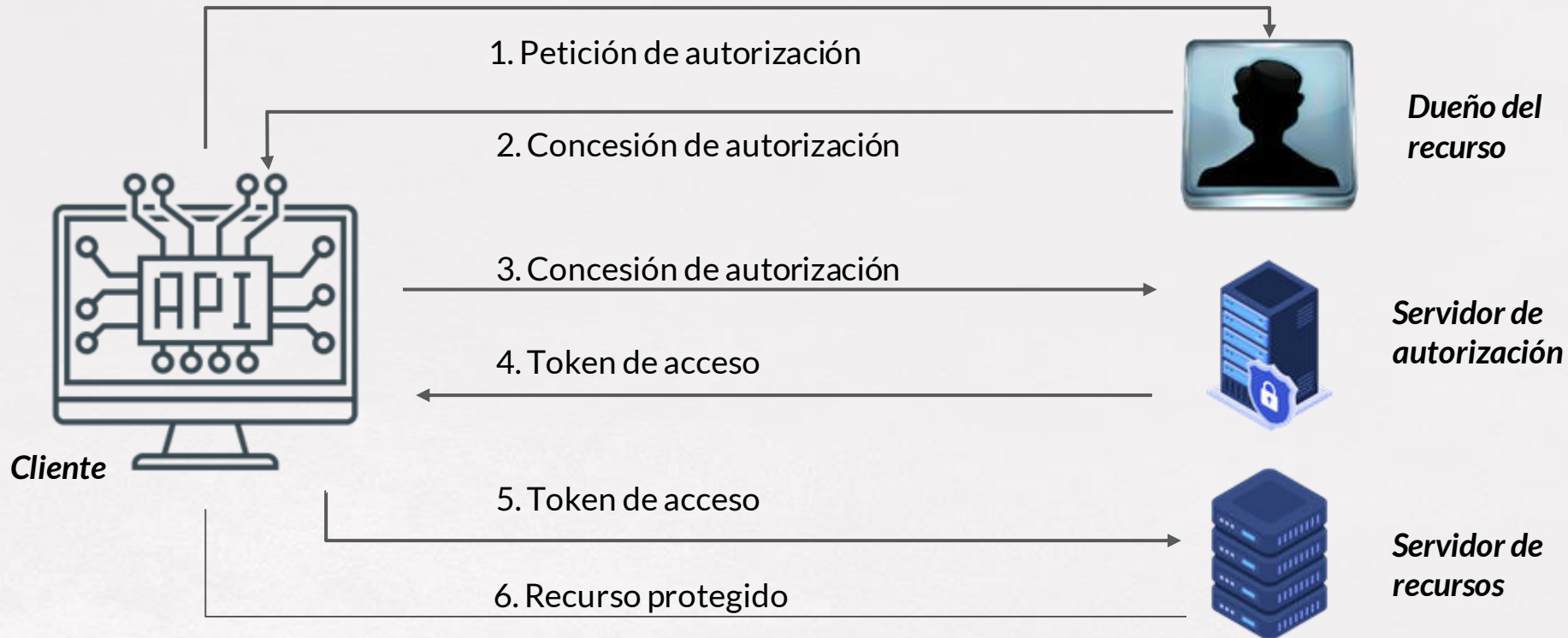
Grant Types o tipos de otorgamiento

- Diferentes formas de obtener el token.
- Surgen a causa de los diferentes tipos de clientes que pueden querer acceder a una serie de recursos.
 - Una aplicación móvil nativa.
 - Una aplicación web con Angular.
 - Una TV con una aplicación en la plataforma X.
 - Un dispositivo IoT como una bombilla inteligente.

Grant Types

- Authorization Code
- Implicit
- Resource Owner Password Credentials
- Client Credentials Flow
- Device Code Flow
- ...

Authorization Code



Authorization Code

- Es el más completo de todos.
- Se utiliza con clientes confidenciales (que son capaces de guardar la contraseña convenientemente).
- Veamos los pasos que se siguen

Authorization Code

- El cliente redirige al usuario al endpoint de autorización, con una serie de parámetros

`https://autorizacion.servidor.com/authorize?response_type=code&client_id=the-client-id&state=xyz&redirect_uri=https://cliente.ejemplo.com/cb&scope=api_read`

- *response_type*: tipo de flujo (code)
- *client_id*: identificador del cliente
- *redirect_uri*: url de vuelta a nuestra aplicación
- *scope*: para qué queremos esta autorización

Authorization Code

- Cuando el cliente es validado, se devuelve una respuesta así:

`https://cliente.ejemplo.com/cb?code=AbCdEfGHiJK12345&state=xyz`

- *code*: código que representa el consentimiento del usuario y su autorización
 - *state*: debe ser igual que en la petición
- Con el código, hacemos una petición POST como la siguiente

Authorization Code

POST /token HTTP/1.1

Host: autorizacion.servidor.com

Authorization: Basic afds8709afs8790asf (client-id:client-secret en base64)

grant_type=authorization_code

&code=AbCdEfGHiJK12345

&redirect_uri=https://cliente.ejemplo.com/cb

Authorization Code (alternativa)

POST /token HTTP/1.1

Host: autorizacion.servidor.com

grant_type=authorization_code

&code=AbCdEfGHiJK12345

&redirect_uri=https://cliente.ejemplo.com/cb

&client_id=the-client-id

&client_secret=qwepuirqewipor09748nmenads

Respuesta (si todo va bien)

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

```
{  
  "access_token": "2YotnFZFEjr1zCsicMWpAA",  
  "token_type": "example",  
  "expires_in": 3600,  
  "refresh_token": "tGzv3JOkF0XG5Qx2TIKWIA",  
  "example_parameter": "example_value"  
}
```

Implicit

- Se utiliza con clientes públicos (que no son capaces de guardar la contraseña convenientemente).
- Pensado para aplicaciones Javascript, Angular, ...
- Veamos los pasos que se siguen

Implicit

- El cliente redirige al usuario al endpoint de autorización, con una serie de parámetros

`https://autorizacion.servidor.com/authorize?response_type=token&client_id=the-client-id&state=xyz&redirect_uri=https://cliente.ejemplo.com/cb&scope=api_read`

- *`response_type`*: token
- *`client_id`*
- *`redirect_uri`*
- *`scope`*

Implicit

- Cuando el cliente es validado, se devuelve una respuesta así:

`https://cliente.ejemplo.com/cb?access_token=ABCDEFdaf379489a&token_type=example&expires_in=3600&state=xyz`

- ***`access_token`***: el token
- ***`token_type`***: tipo de token
- ***`expires_in`***: tiempo de vida
- ***`state`***: debe ser igual que en la petición

Password

- Apropiado cuando entre el cliente y el servidor de autorización hay una relación de confianza.
- Debería ser usado cuando no se pueda utilizar otra alternativa de flujo.
- Se puede utilizar para migrar desde la autenticación Básica hacia OAuth2

Password

POST /token HTTP/1.1

Host: autorizacion.servidor.com

Authorization: Basic afds8709afs8790asf (client-id:client-secret en base64)

Content-Type: application/x-www-form-urlencoded

grant_type=password

&username=luismi

&password=AsDf1234

Respuesta (si todo va bien)

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

```
{  
  "access_token": "2YotnFZFEjr1zCsicMWpAA",  
  "token_type": "example",  
  "expires_in": 3600,  
  "refresh_token": "tGzv3JOkF0XG5Qx2TIKWIA",  
  "example_parameter": "example_value"  
}
```

Client Credentials

- Apropiado cuando no existen usuarios propietarios del recurso. Es decir, si no hay usuarios involucrados.
- Sin haberlos, podemos seguir utilizando OAuth para proteger nuestra API.
- La aplicación cliente, en sí, es el propietario del recurso y no hay usuarios involucrados.

Client credentials

POST /token HTTP/1.1

Host: autorizacion.servidor.com

Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials

&client_id=the-client-id

&client_secret=qwepuirqewipor09748nmenads

&scope=API_READ

Respuesta (si todo va bien)

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

```
{  
  "access_token": "2YotnFZFEjr1zCsicMWpAA",  
  "token_type": "example",  
  "expires_in": 3600,  
  "refresh_token": "tGzv3JOkF0XG5Qx2TIKWIA",  
  "example_parameter": "example_value"  
}
```

OAuth2: Servidor de autenticación

Implementa la seguridad de tu API Rest con Spring Boot

Spring Security y OAuth2

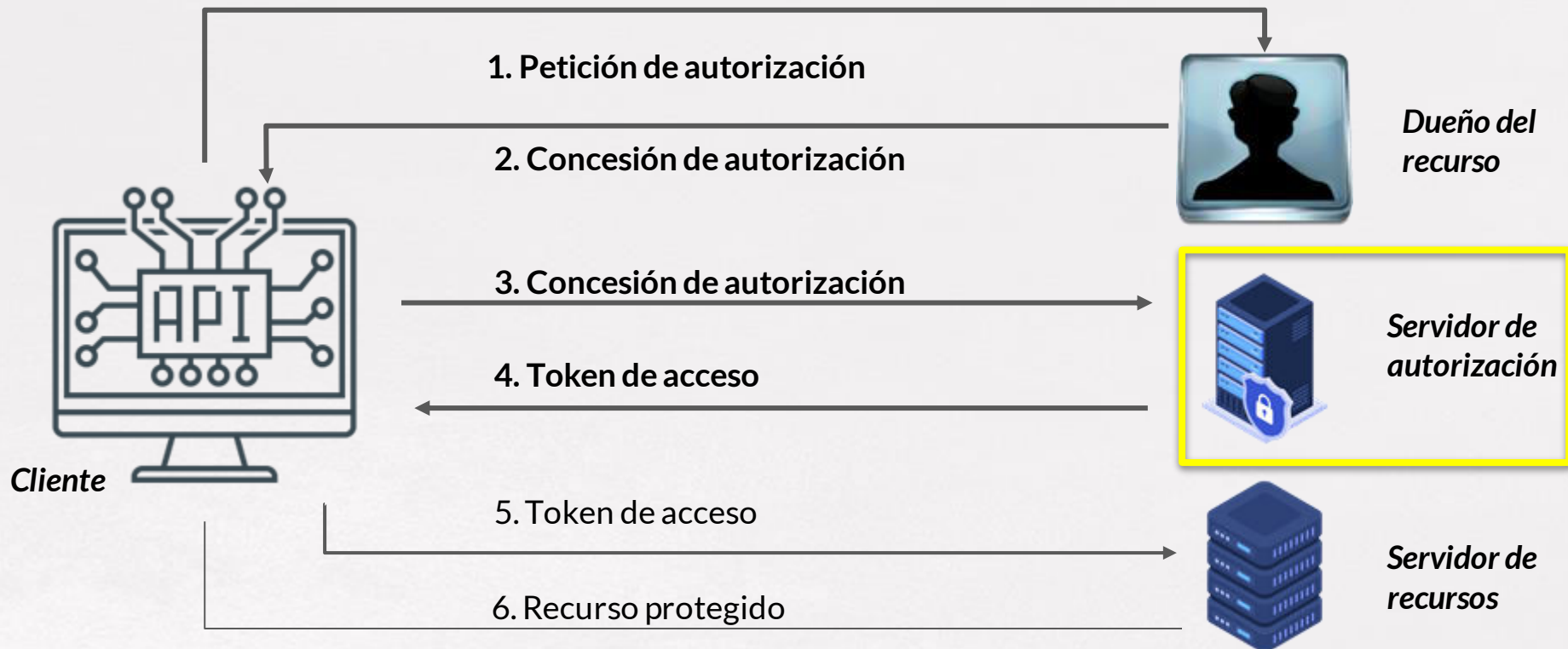
- A lo largo de la versión 5 (5.0, 5.1, 5.2) la implementación de OAuth2 en Spring Security está sufriendo cambios.
- Se puede encontrar una matriz con este cambio en <https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Features-Matrix>
- Vamos a realizar un ejemplo que nos pueda dar garantías tanto con respecto a proyectos *legacy* como con respecto al futuro.

Spring Boot y OAuth2

- En versiones anteriores de Spring Boot, existía una integración automática.
- A partir de la versión 2, OAuth2 queda dentro de Spring Security.
- Si queremos utilizar algunos componentes, como el servidor de autorización, tenemos que añadir una dependencia.

```
<groupId>org.springframework.security.oauth.boot</groupId>  
    <artifactId>spring-security-oauth2-autoconfigure</artifactId>  
<version>2.X.Y.RELEASE</version>
```

¿En qué parte nos centramos en este momento?



Servidor de autorización

- Extiende a *AuthorizationServerConfigurerAdapter*
- Anotado con *@EnableAuthorizationServer* + *@Configuration*
- Configuramos
 - Los diferentes clientes con sus características
 - La seguridad de los tokens
 - Conexión con el modelo de autenticación
 -

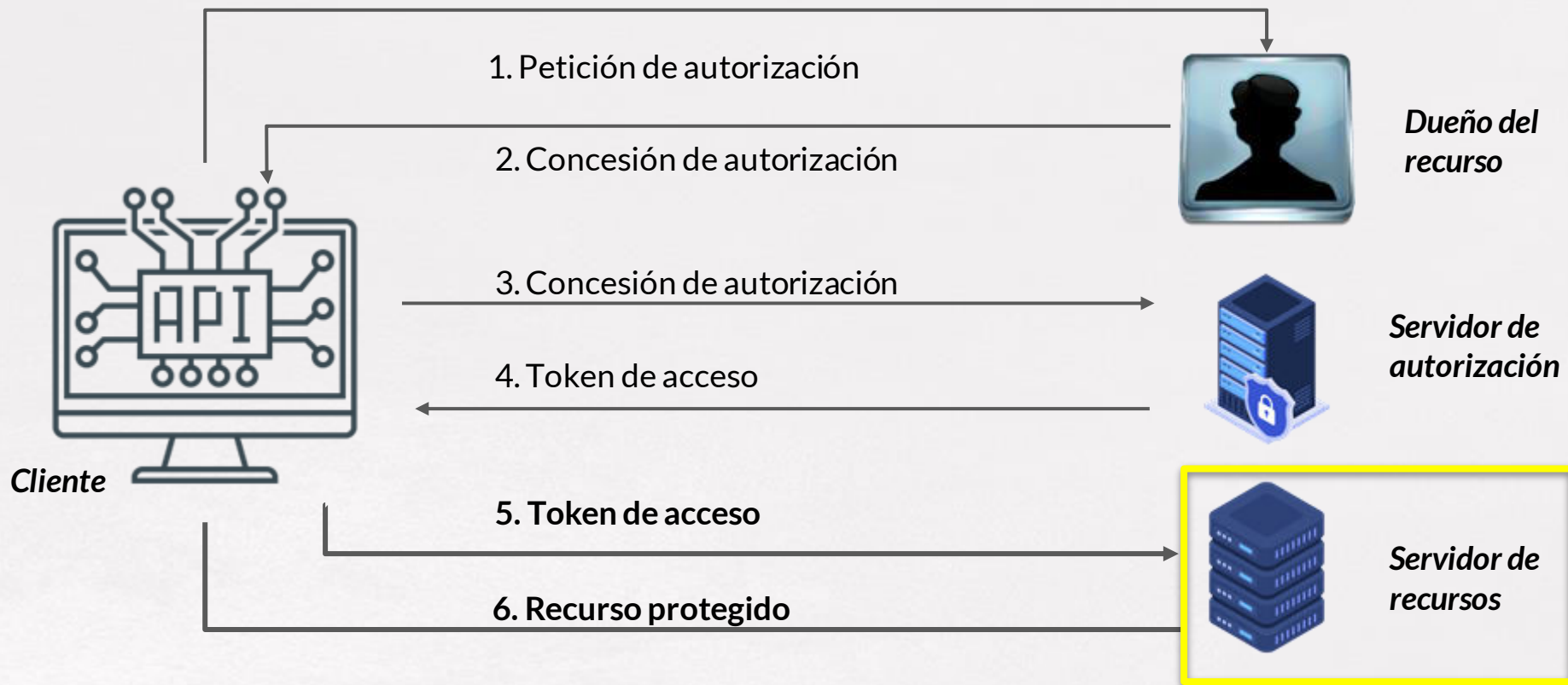
Cambios en *SecurityConfig*

- Exponemos el método de autenticación como un bean
- Permitimos acceder al login (el login por defecto, será el que utilizemos como login del servidor de autorización).
- Modificamos la precedencia del bean, dándole más que sobre el resto del mismo tipo.
- Comprobaremos el funcionamiento de todo más adelante.

OAuth2: servidor de recursos

Implementa la seguridad de tu API Rest con Spring Boot

¿En qué parte nos centramos en este momento?



Servidor de recursos

- Es el encargado de proteger los recursos de nuestro sistema
- Comprueba, gracias al servidor de autorización, los tokens que recibe en las peticiones, para verificar si está autenticado.
- Se parece mucho a las clases de configuración de seguridad que hemos trabajado en lecciones anteriores.

Servidor de autorización

- En el podemos configurar
 - La protección de los recursos
 - Por defecto, todo lo que no esté en /oauth/** está protegido, pero sin reglas específicas.
 - Propiedades específicas del servidor de recursos (como *resource id*).

Reconfigurando CORS

Implementa la seguridad de tu API Rest con Spring Boot

Algunos cambios

- Para que OAuth2 funcione correctamente (cualquier tipo de *grant type*) necesitamos hacer algunos cambios en la seguridad.

CORS

- Eliminamos la configuración heredada del proyecto base.
- Configuramos a través de una instancia de Filter.
- Le damos la prioridad más alta a dicho filtro.

SecurityConfig

- Permitimos que se realicen peticiones OPTIONS a cualquier URL.
- Esto suele ser necesario para algunos tipos de clientes, como Angular.
- También añadimos otros elementos de configuración.

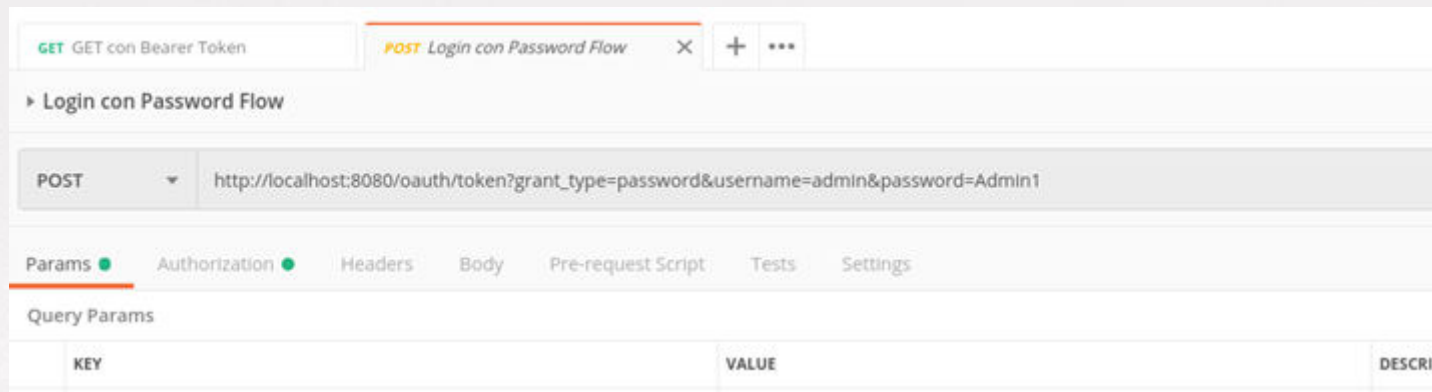
Ya estamos listos para probar
nuestra aplicación (o casi :S)

OAuth2: Ejecución de nuestra aplicación

Implementa la seguridad de tu API Rest con Spring Boot

Pruebas con Postman

- Probamos el flujo de tipo Password
- *client_id* y *client_secret* como autenticación básica



Pruebas con Postman

- Podemos usar el Bearer token para hacer otras peticiones

```
1 {  
2   "access_token": "517d0695-3ed4-4407-b3b0-f96a9e4f513d",  
3   "token_type": "bearer",  
4   "refresh_token": "5119940f-f0ec-4b73-991e-df522217fa06",  
5   "expires_in": 84862,  
6   "scope": "read"  
7 }
```

Pruebas con Postman

GET <http://localhost:8080/producto/5>

TYPE
Bearer Token

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

[Preview Request](#)

Token
517d0695-3ed4-4407-b3b0-f96a9e4f513d


Body Cookies (1) Headers (13) Test Results Status: 200 OK Time

Pretty Raw Preview Visualize **BETA** JSON

```
1 {  
2   "id": 5,  
3   "nombre": "Wine - Beringer Founders Estate",  
4   "precio": 27.0,  
5   "imagen": "http://dummyimage.com/139x103.bmp/5fa2dd/ffffff",  
6   "categoria": {  
7     "id": 2,  
8     "nombre": "Bebida"  
9   },  
10  "lotes": []  
11 }
```

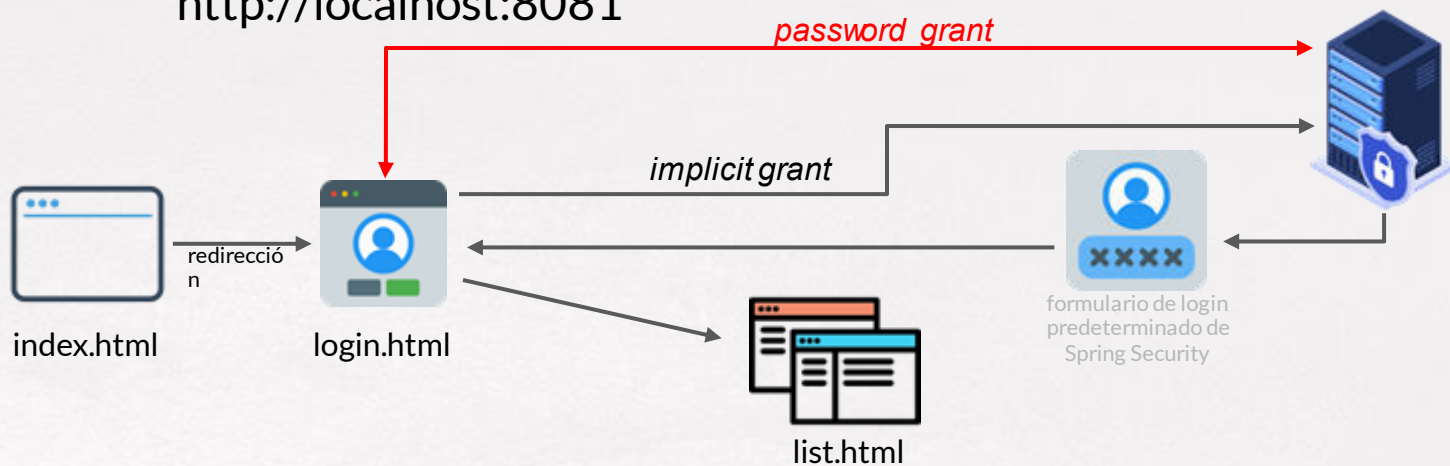
Pruebas con Postman

- Hemos añadido el controlador `/user/me` de otros proyectos de ejemplo, que nos da la información del perfil de usuario en función de su token.

```
Pretty Raw Preview Visualize BETA JSON 
1 {
2   "username": "admin",
3   "avatar": "https://api.adorable.io/avatars/285/admin@openwebinars.net.png",
4   "fullName": "Admin admin",
5   "email": "admin@openwebinars.net",
6   "roles": [
7     "ADMIN",
8     "USER"
9   ]
10 }
```

Aplicación cliente

- Probamos la implementación de un cliente Javascript muy sencillo que nos permita probar el flujo *implícito* y el *password*.
- Ejecutamos el proyecto y cargamos en el navegador `http://localhost:8081`



OAuth2: Tokens en base de datos

Implementa la seguridad de tu API Rest con Spring Boot

Hasta ahora

- Nuestros clientes se han almacenado en memoria

```
public void configure(ClientDetailsServiceConfigurer clients) {  
    clients  
        .inMemory()
```

- También nuestros tokens
 - Por defecto, se configura en memoria (*InMemoryTokenStore*)

¿Y si queremos almacenar en base de datos?

- Necesitamos proporcionar un esquema para
 - clientes
 - tokens
 - códigos de autorización
 - ...
- Tenemos uno en el repositorio oficial en github:

<https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql>

import.sql

- Si queremos crear algunas tablas a través de un script DDL, y mantener la creación automática de las tablas asociadas a entidades y asociaciones, podemos crear un fichero llamado **import.sql**.
- En producción
 - Posiblemente utilizemos `ddl-auto=none`
 - También algún sistema de migración de versiones de bases de datos, como *Liquibase* o *Flyway*

Configuración de H2

- Para tener una base de datos persistente.
- También para poder consultar a través de la consola.
 - Puntualmente, configuramos la seguridad para poder acceder a la consola.

Servidor de autorización

- Inyectamos el datasource (configurado vía *properties*).
- Configuramos los clientes a través de JDBC

```
public void configure(ClientDetailsServiceConfigurer clients) {  
    clients.jdbc(dataSource) ...  
}
```

- Creamos un TokenStore almacenado a través de JDBC, y lo configuramos.

```
@Bean  
public TokenStore tokenStore() {  
    return new JdbcTokenStore(dataSource);  
}
```

Ejecución

- Podemos probar a solicitar un token con POSTMAN.
- Si el mismo usuario vuelve a loguearse, le devuelve el mismo token.
- Podemos ejecutar la consola de H2 y verificar que los tokens están allí.

OAuth2 con Json Web Token

Implementa la seguridad de tu API Rest con Spring Boot

Json Web Token

- JSON Web Token (RFC 7519)
- Es un mecanismo para propagar de forma segura la identidad (y *claims* o privilegios) entre dos partes.
- Los privilegios se codifican como objetos JSON.
- *Access Token vs. JWT Token.*
 - Los tokens JWT pueden incluir información que pueden minimizar peticiones.
 - Por ejemplo: GET */user/me* vs. extraer información del token.

AccessTokenConverter

- Interfaz que permite almacenar la información de autenticación dentro del Token.
- Varias implementaciones, entre ellas *JwtAccessTokenConverter*.
- traduce entre tokens JWT e información de autenticación OAuth (en ambas direcciones).
- Actúa también como *TokenEnhancer*, los cuales permiten mejorar un token antes de almacenarlo o enviarlo al cliente.

Ejecución

- Podemos probar a solicitar un token con POSTMAN.
- El token tendrá estructura JWT
- Si el mismo usuario vuelve a loguearse, le devuelve el mismo token, pero JWT.
- Si revisamos desde JWT debugger, el token incluye información del perfil de usuario.