

Instructor Luis
Márquez

Master Definitivo: Spring Security 6, JWT y OAuth2

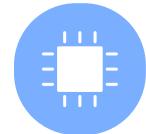
Conviértete en un experto en seguridad web con
Spring Security 6, JWT y OAuth2! Aprende a
proteger tus aplicaciones, gestionar roles y permisos
con Spring Boot 3 en este curso completo y práctico





Sección 01: Introducción





¿Qué vamos a aprender?



Conceptos de spring security 6 y JWT desde cero



Arquitectura de spring security



Definir un sistema de roles y permisos dinámico



Manejar excepciones de seguridad



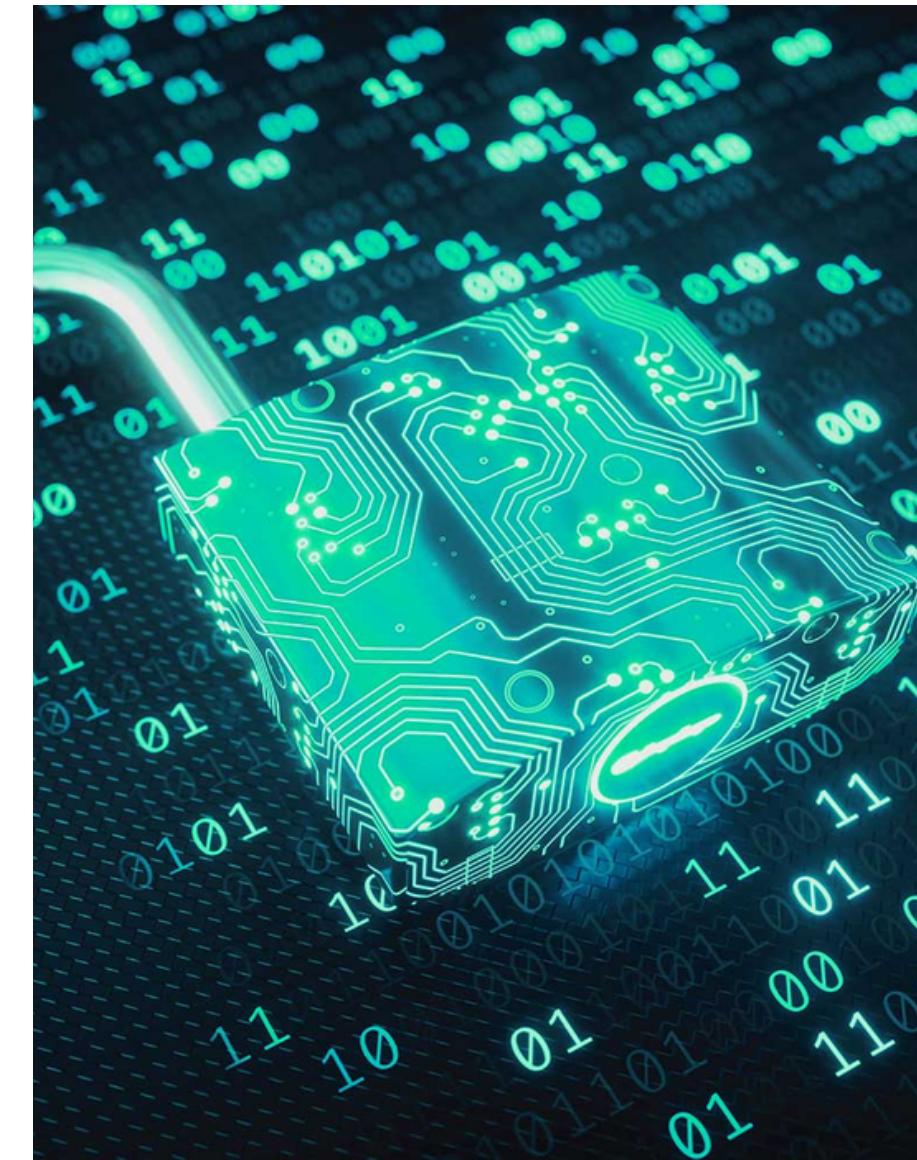
Protocolo OAuth2 desde cero



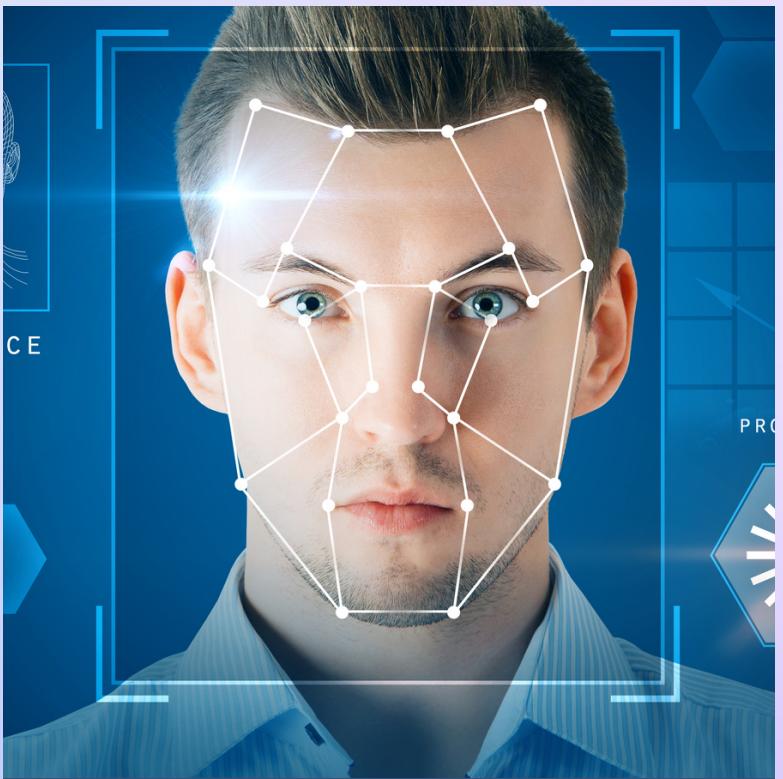
Crear arquitecturas de seguridad híbridas

Spring security 6

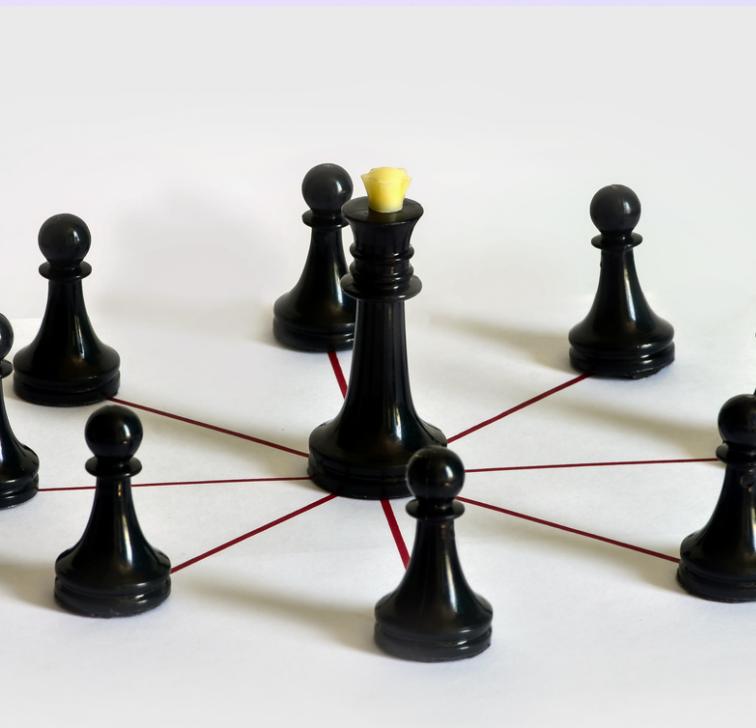
Spring Security: Es un módulo del ecosistema de spring que protege tus aplicaciones web y APIs con autenticación y autorización segura.



Puntos clave de Spring Security



Autenticación
¿Quién soy?



Autorización
¿Sobre qué tengo
permisos?



**Protección de
recursos**
¿Qué recursos son
públicos y cuáles son
protegidos?



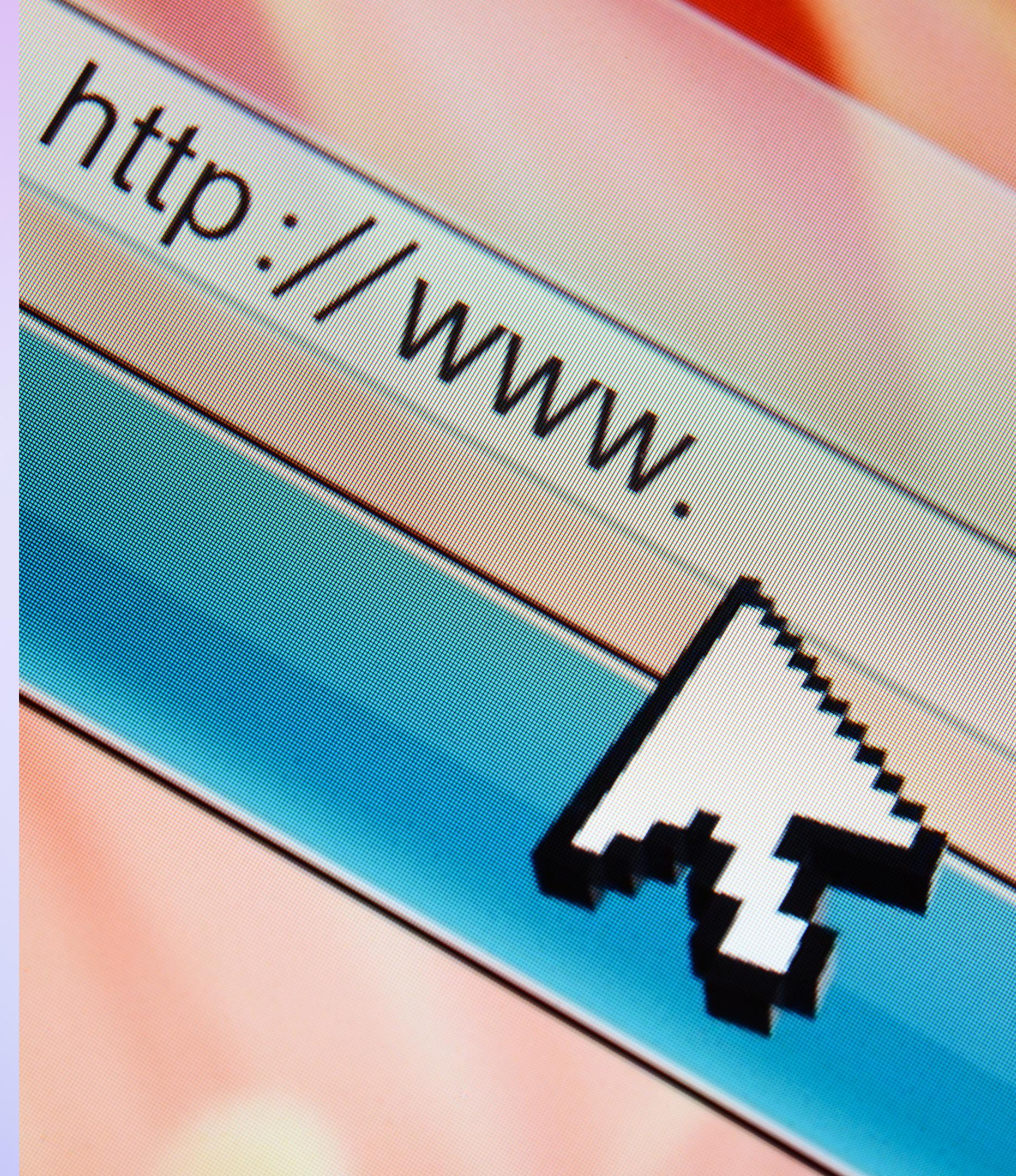
**Integración con
diferentes mecanismos
de autenticación**
¿Cómo hago la autenticación
y autorización ?



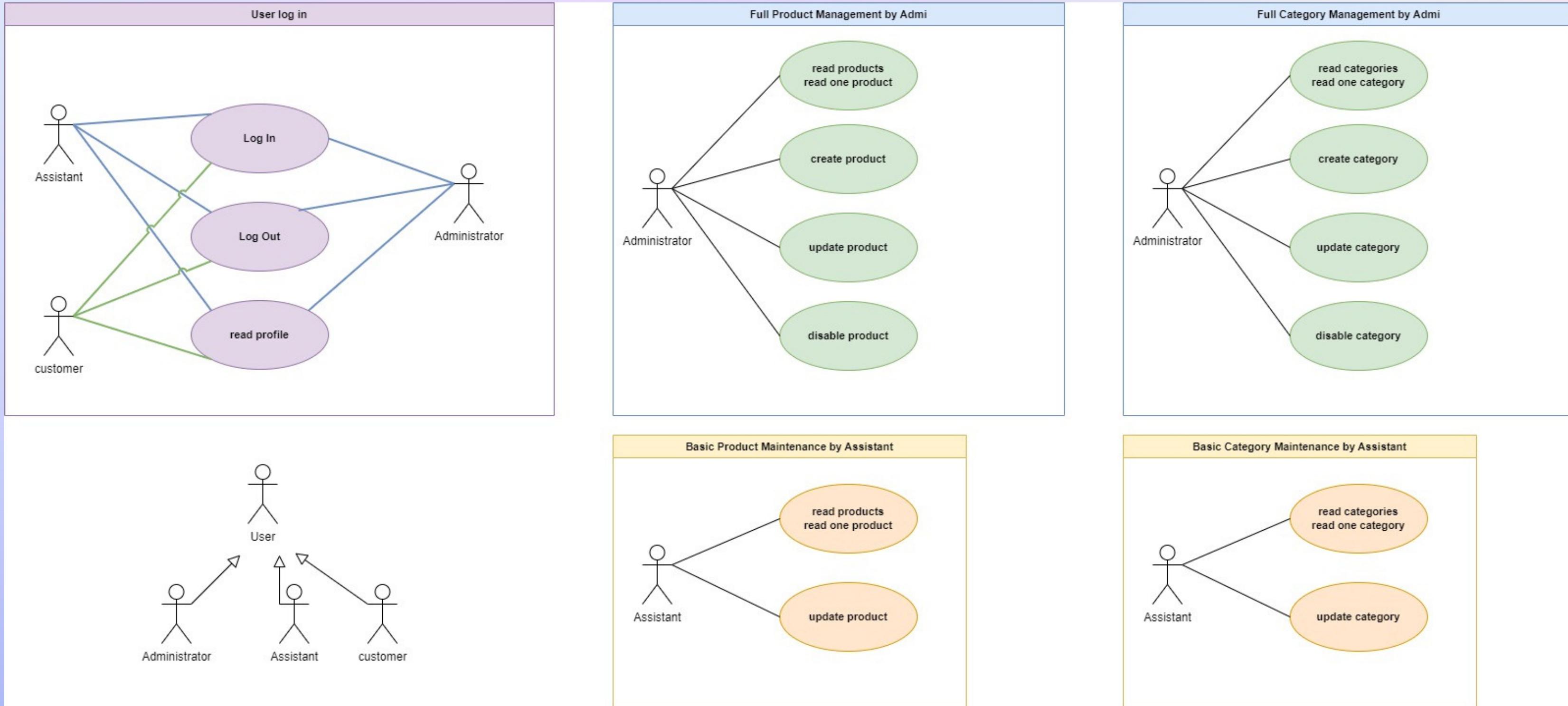
¿Por qué es importante Spring Security?

La seguridad es un aspecto crítico en cualquier aplicación web o API.

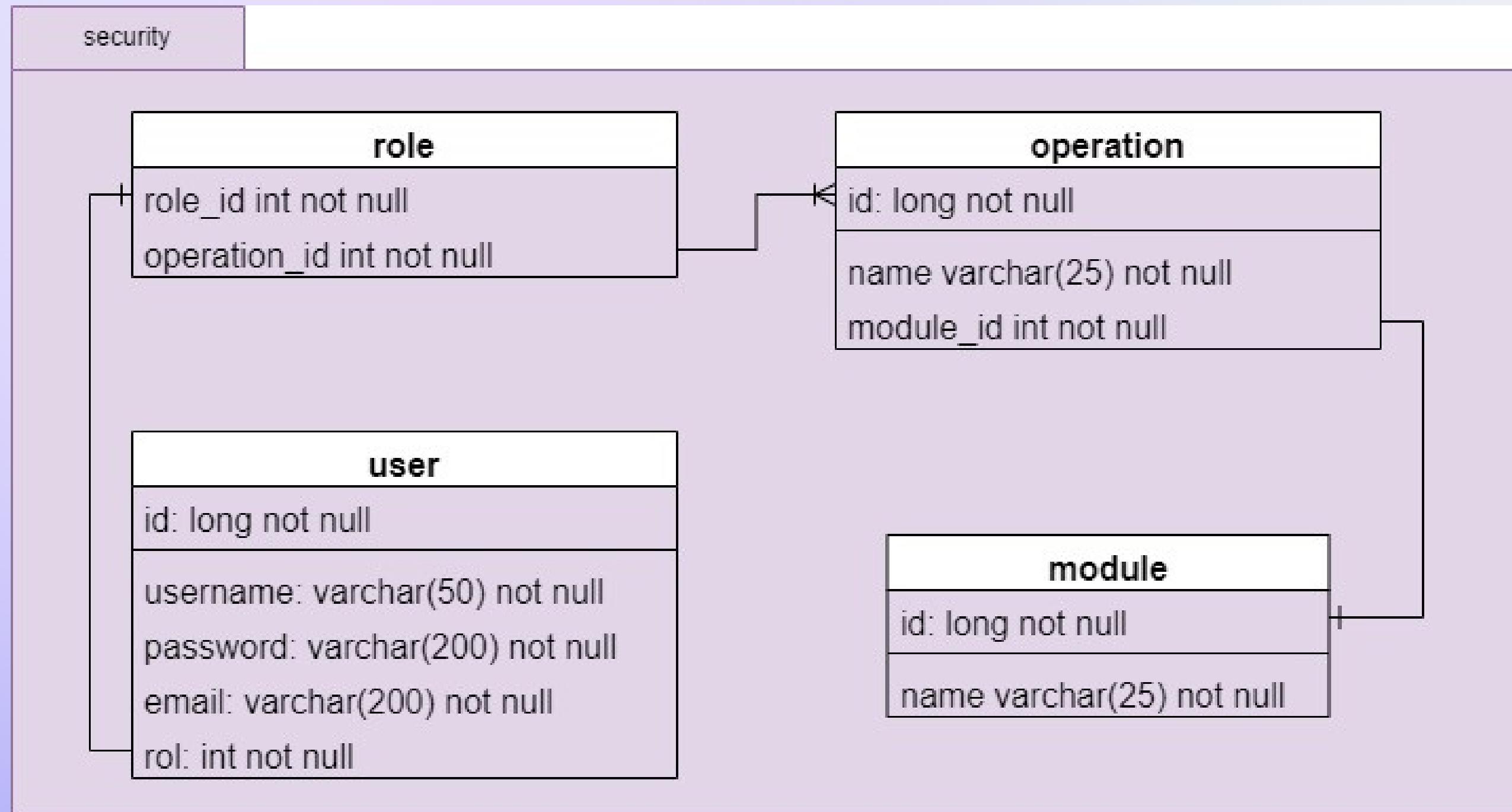
Spring Security nos proporciona una solución robusta y confiable para implementar medidas de seguridad en nuestras aplicaciones sin tener que preocuparnos por detalles complicados.



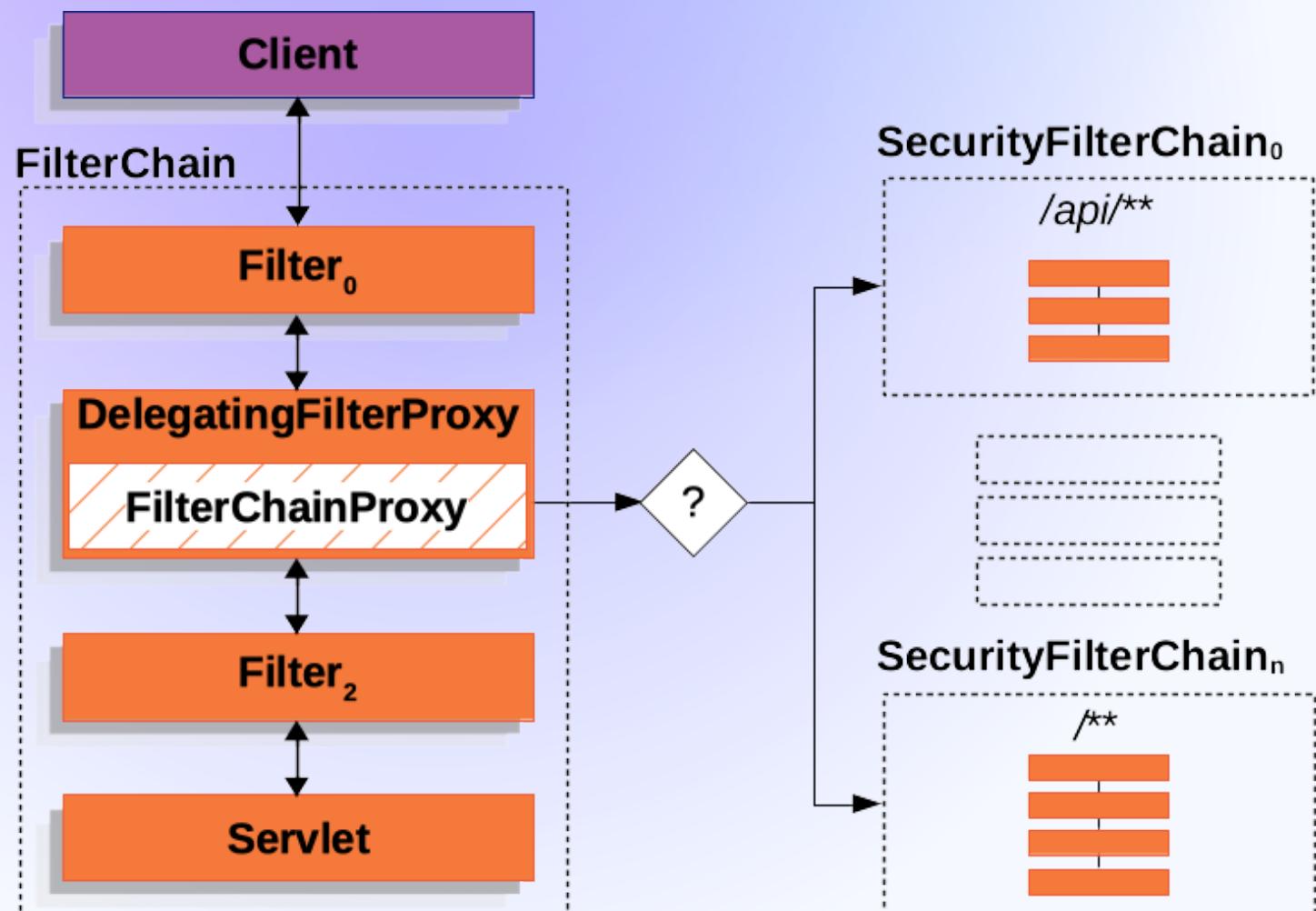
¿Qué vamos a construir?



¿Qué vamos a construir?



Sección 02: Spring Security 6 y JWT el panorama



Explora las novedades de Spring Security 6 y la potencia de JSON Web Tokens (JWT)

- ✓ **Spring security y su arquitectura de alto nivel**
- ✓ **Qué es JWT y su importancia**

Arquitectura a gran escala

DelegatingFilterProxy

Es una clase de Spring Framework que actúa como un delegado para un filtro definido en el contexto de la aplicación. En Spring Security, se utiliza principalmente para integrar la cadena de filtros de seguridad de Spring Security con la configuración de filtros de una aplicación web basada en Servlet.

FilterChainProxy

Es el componente central de Spring Security que maneja la coordinación de la cadena de filtros de seguridad para proteger una aplicación web.

SecurityFilterChain

Es una interfaz que representa una cadena de filtros de seguridad que se aplican a las solicitudes HTTP en una aplicación web. Esta interfaz juega un papel crucial en la configuración y el funcionamiento de la seguridad en Spring Security.

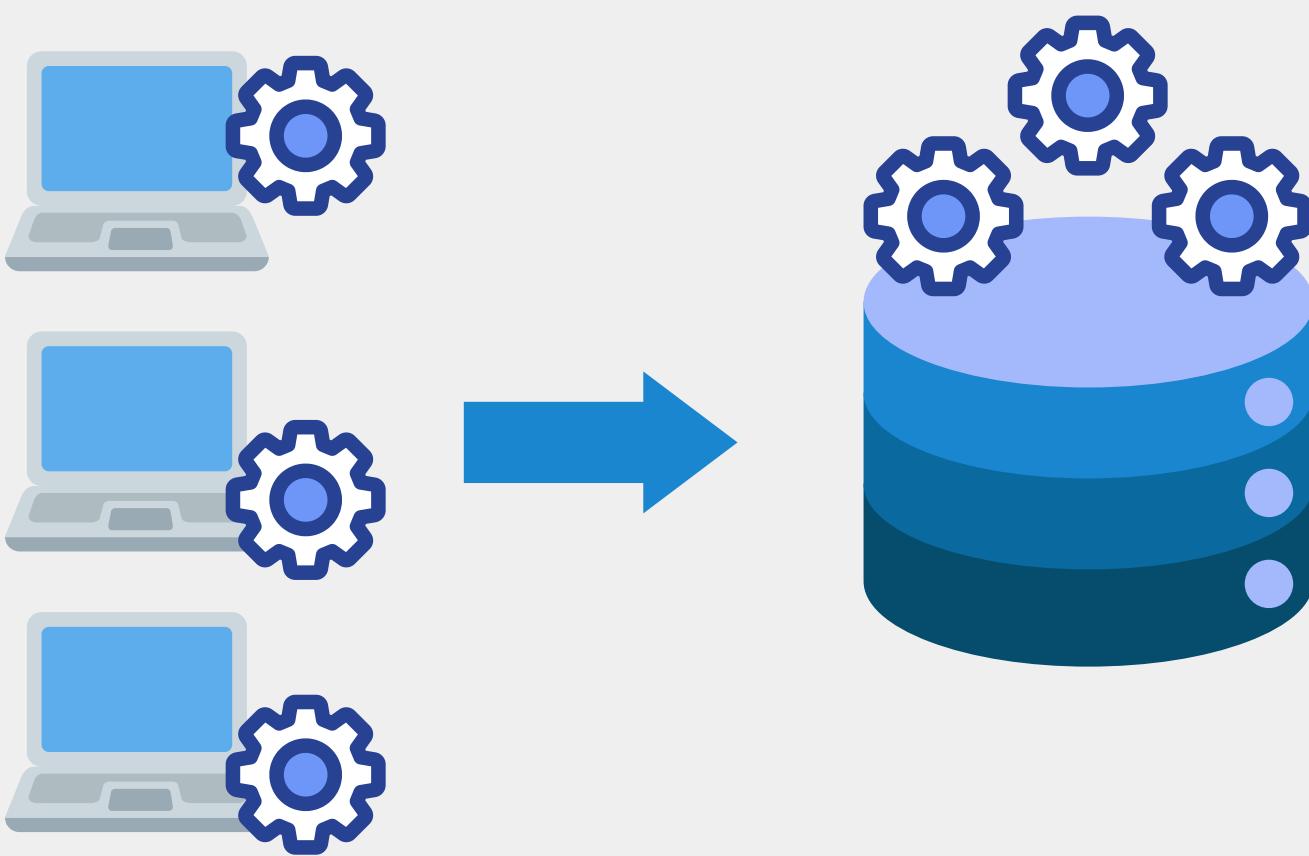
Aplicaciones stateless vs aplicaciones stateful





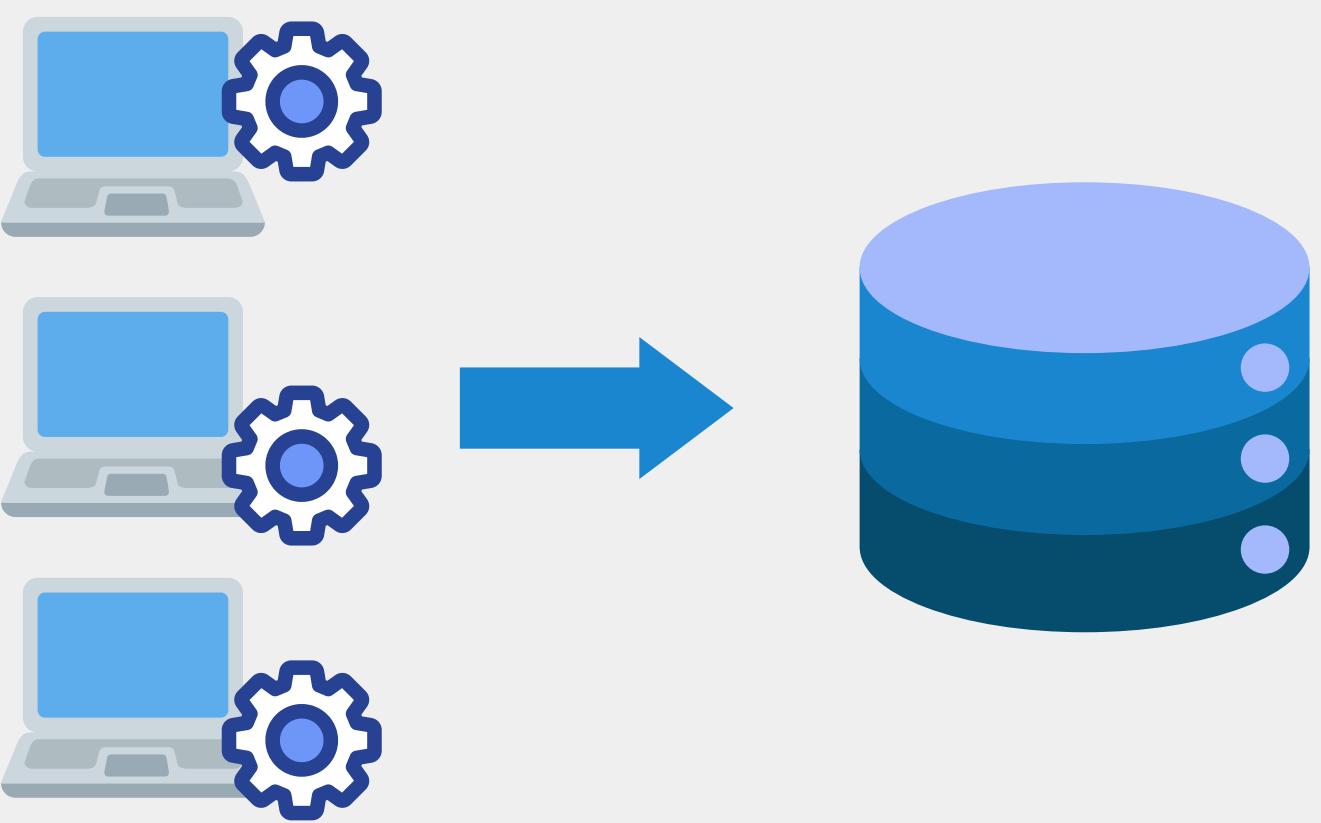
stateful

(Seguridad basada en sesiones)



stateless

(Seguridad basada en tokens de autenticación)





stateful

(Seguridad basada en sesiones)

Implica mantener un estado en el servidor para cada usuario que interactúa con la aplicación.



Eficaz para mantener información del usuario en el servidor y administrar su sesión.



Requiere el almacenamiento y la gestión de sesiones en el servidor, lo que puede generar problemas de escalabilidad y rendimiento en aplicaciones con muchos usuarios concurrentes.

stateless

(Seguridad basada en tokens de autenticación)

Se basa en la idea de que cada solicitud que realiza el cliente contiene toda la información necesaria para que el servidor la procese



Altamente escalable y eficiente, ya que el servidor no necesita almacenar información del usuario.



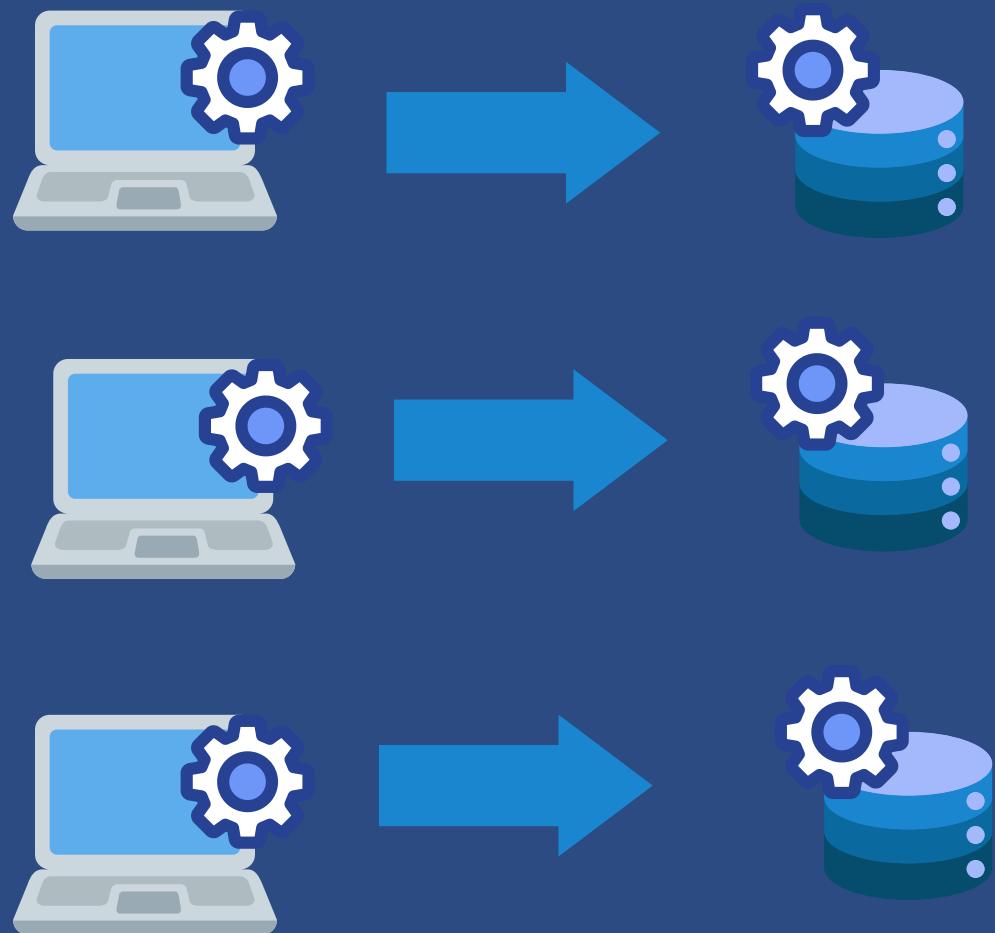
Cada solicitud se procesa de manera independiente



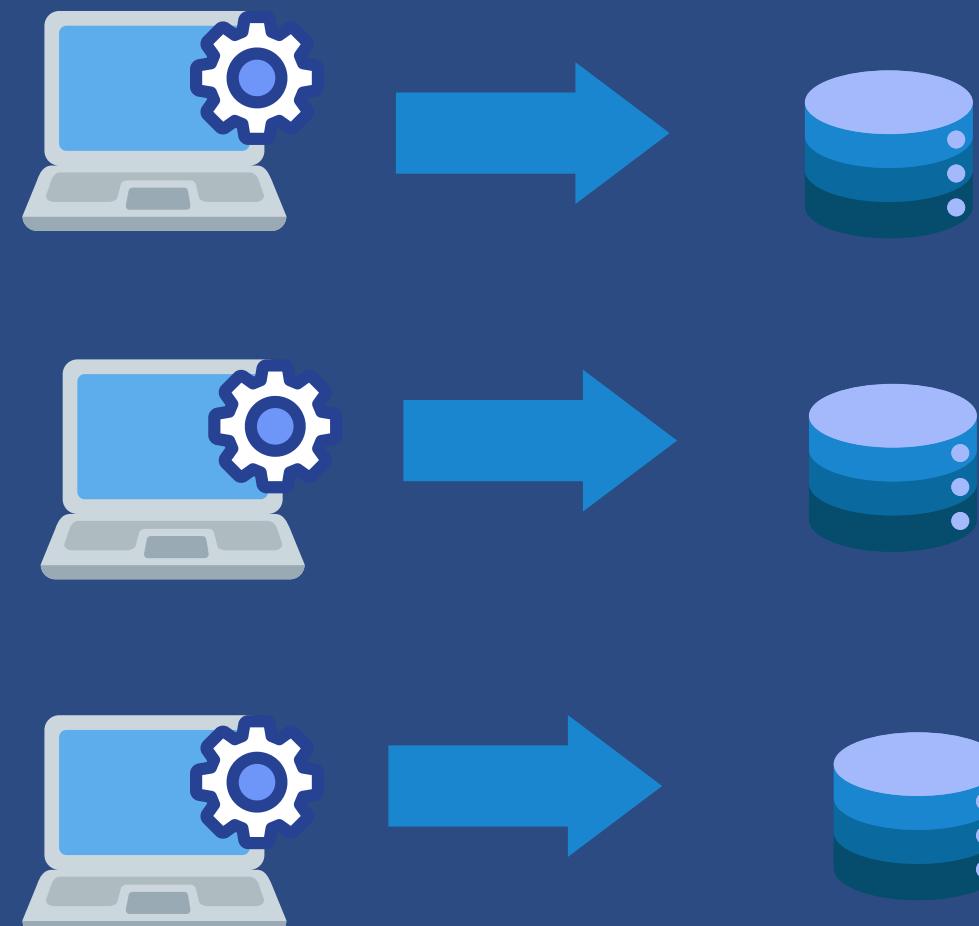
Como toda la información necesaria se incluye en cada solicitud, los tokens o JWT deben ser protegidos adecuadamente para evitar accesos no autorizados

Escalabilidad

stateful



stateless





JWT: Json Web Token

Define un método compacto y seguro para transmitir información entre dos partes en formato JSON

Está diseñado para ser utilizado en entornos donde la confianza entre las partes es esencial, como autenticación y autorización en aplicaciones web y APIs.

Partes de un JWT

Un JWT consta de tres partes separadas por puntos: el encabezado, la carga útil y la firma.

- El encabezado contiene información sobre el tipo de token y el algoritmo de firma utilizado.
- La carga útil contiene los datos que queremos transmitir, como el nombre de usuario o los roles del usuario.
- Y la firma es una cadena codificada que garantiza que el token no ha sido modificado durante su transmisión



Ejemplo

1 - Un cliente hace login y el servidor le genera un token firmado con una clave segura. Este token tiene un role de tipo CUSTOMER

POST localhost:9191/api/v1/auth/authenticate

Params Authorization Headers (8) Body Pre-req

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

```
1 {  
2   "username": "lmarquez",  
3   "password": "clave123"  
4 }
```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzIE3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJyb2xIjoiUk9MRV9DVVNUT01FUiJ9.5qVJLM6XWijmFcZgqtXUFR0eptqAZFKC6ycsGzYtuKQ

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

PAYOUT: DATA

```
"sub": "lmarquez",  
"name": "Luis Márquez",  
"iat": 1516239022,  
"role": "ROLE_CUSTOMER"
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  mi clave secreta  
)  secret base64 encoded
```

Ejemplo

2 - El cliente decodifica el base64, modifica su rol a ROLE_ADMIN y intenta hacer una solicitud a un recurso protegido para administradores. Al intentar firmar el token modificado no logra dar con la clave segura y lo firma con un texto aleatorio

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ  
JzdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzM  
E3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJy  
b2x1IjoiUk9MRV9BRE1JTiJ9.Qe4Fgfh-  
rE_aWWinU3iyNEmH4AmIyMj0j_SUIpFlHrU
```

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}  
  
PAYLOAD: DATA  
  
{  
  "sub": "lmarquez",  
  "name": "Luis Márquez",  
  "iat": 1516239022,  
  "role": "ROLE_ADMIN"  
}  
  
VERIFY SIGNATURE  
  
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  otra clave secreta  
)  secret base64 encoded
```

Ejemplo

3 - Cuando el backend recibe el nuevo token alterado, toma solo el header y el payload para generar una nueva firma en base al ese header y payload enviados.

El servidor compara la firma genera por él mismo y la compara con la firma enviada y ve que no coinciden entonces deniega el acceso al sistema.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzIE3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJyb2x1IjoiUk9MRV9BRE1JTiJ9.Qe4Fgfh-rE_aWWinU3iyNEmH4AmIyMj0j_SUIpFlHrU

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzIE3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJyb2x1IjoiUk9MRV9BRE1JTiJ9.Qe4Fgfh-rE_aWWinU3iyNEmH4AmIyMj0j_SUIpFlHrU

PAYLOAD: DATA

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

PAYLOAD: DATA

{
  "sub": "lmarquez",
  "name": "Luis Márquez",
  "iat": 1516239022,
  "role": "ROLE_ADMIN"
}

VERIFY SIGNATURE

HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  otra clave secreta
) □ secret base64 encoded
```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzIE3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJyb2x1IjoiUk9MRV9BRE1JTiJ9.Gv5jmVQ4kXullVcpkGTd8tzidN-AkiPfM9cw0yQTjjc

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiJsbWFycXVleiIsIm5hbWUiOiJMdWlzIE3DoXJxdWV6IiwiaWF0IjoxNTE2MjM5MDIyLCJyb2x1IjoiUk9MRV9BRE1JTiJ9.Gv5jmVQ4kXullVcpkGTd8tzidN-AkiPfM9cw0yQTjjc

PAYLOAD: DATA

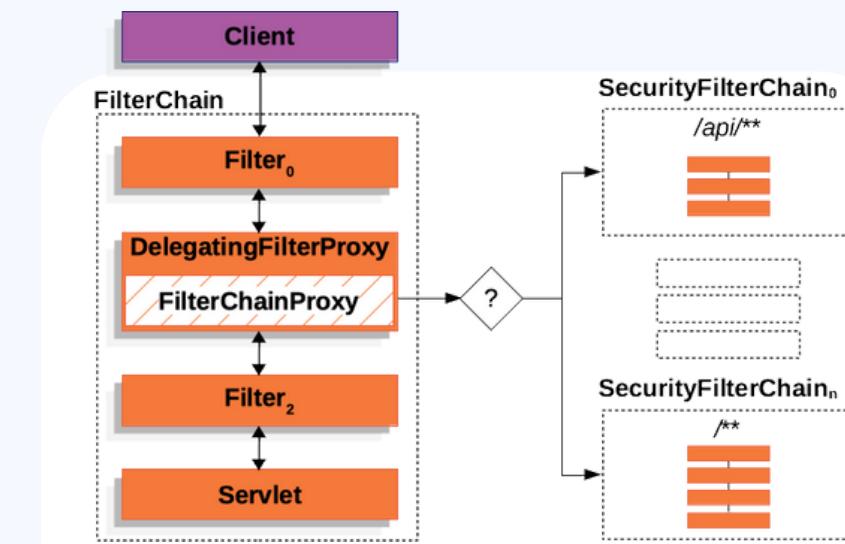
{
  "sub": "lmarquez",
  "name": "Luis Márquez",
  "iat": 1516239022,
  "role": "ROLE_ADMIN"
}

PAYLOAD: DATA

{
  "sub": "lmarquez",
  "name": "Luis Márquez",
  "iat": 1516239022,
  "role": "ROLE_ADMIN"
}

VERIFY SIGNATURE

HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  mi clave secreta
) □ secret base64 encoded
```



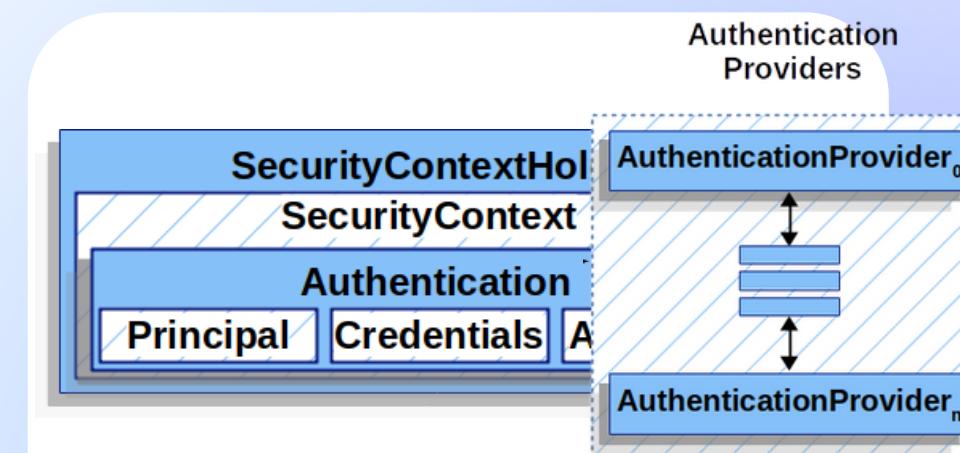
Arquitectura de alto
nivel



Autenticación vrs
autorización

Sección 04: Spring Security

6: arquitectura a fondo



Arquitectura de
autenticación

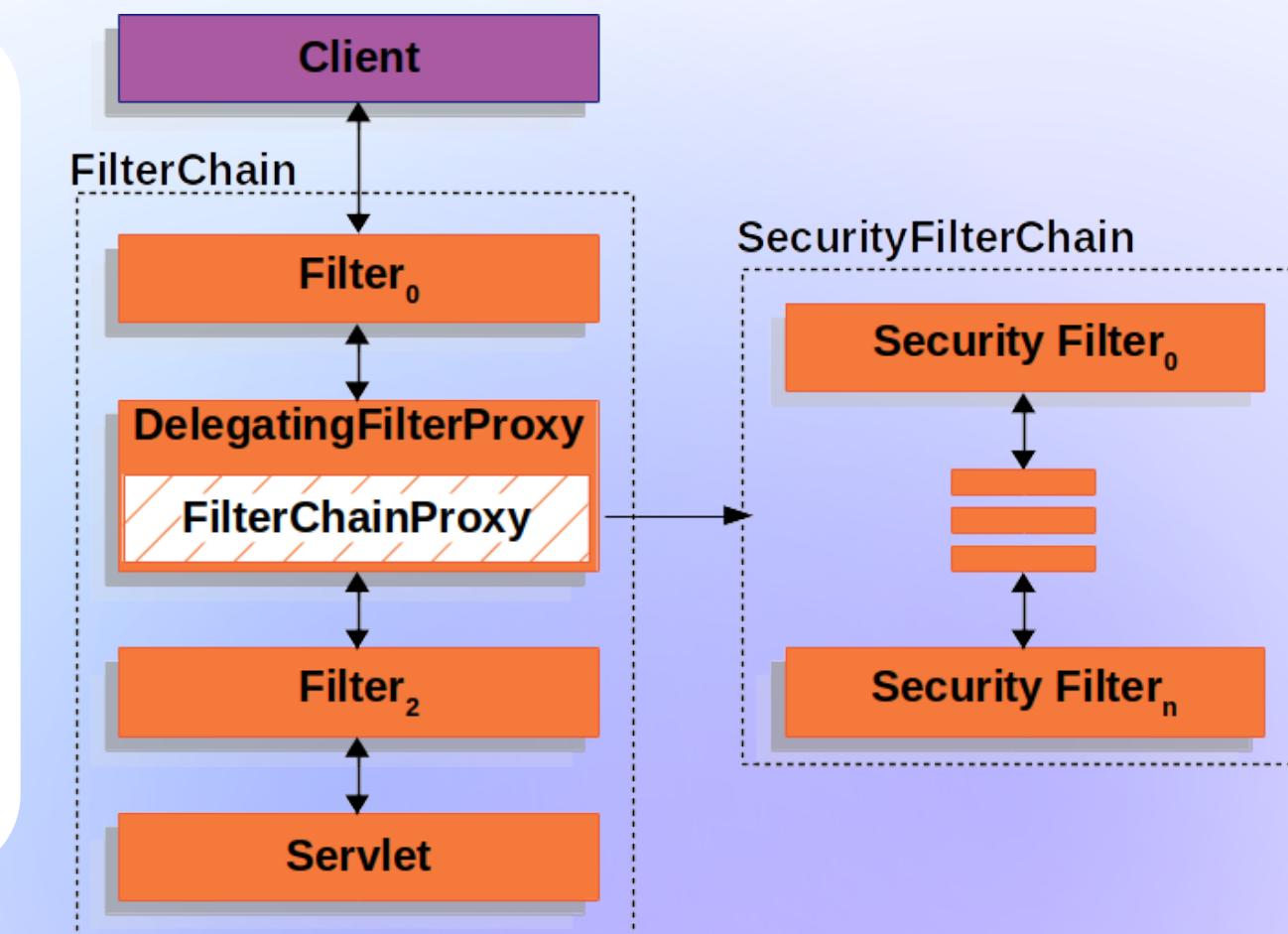
Arquitectura de Spring Security dentro de las aplicaciones basadas en Servlets

Su principal propósito es integrar filtros de aplicación personalizados, incluidos los filtros de seguridad de Spring Security, en el ciclo de vida de una aplicación web basada en Servlet.

— **DelegatingFilterProxy**

Es el componente central de Spring Security que maneja la coordinación de la cadena de filtros de seguridad para proteger una aplicación web.

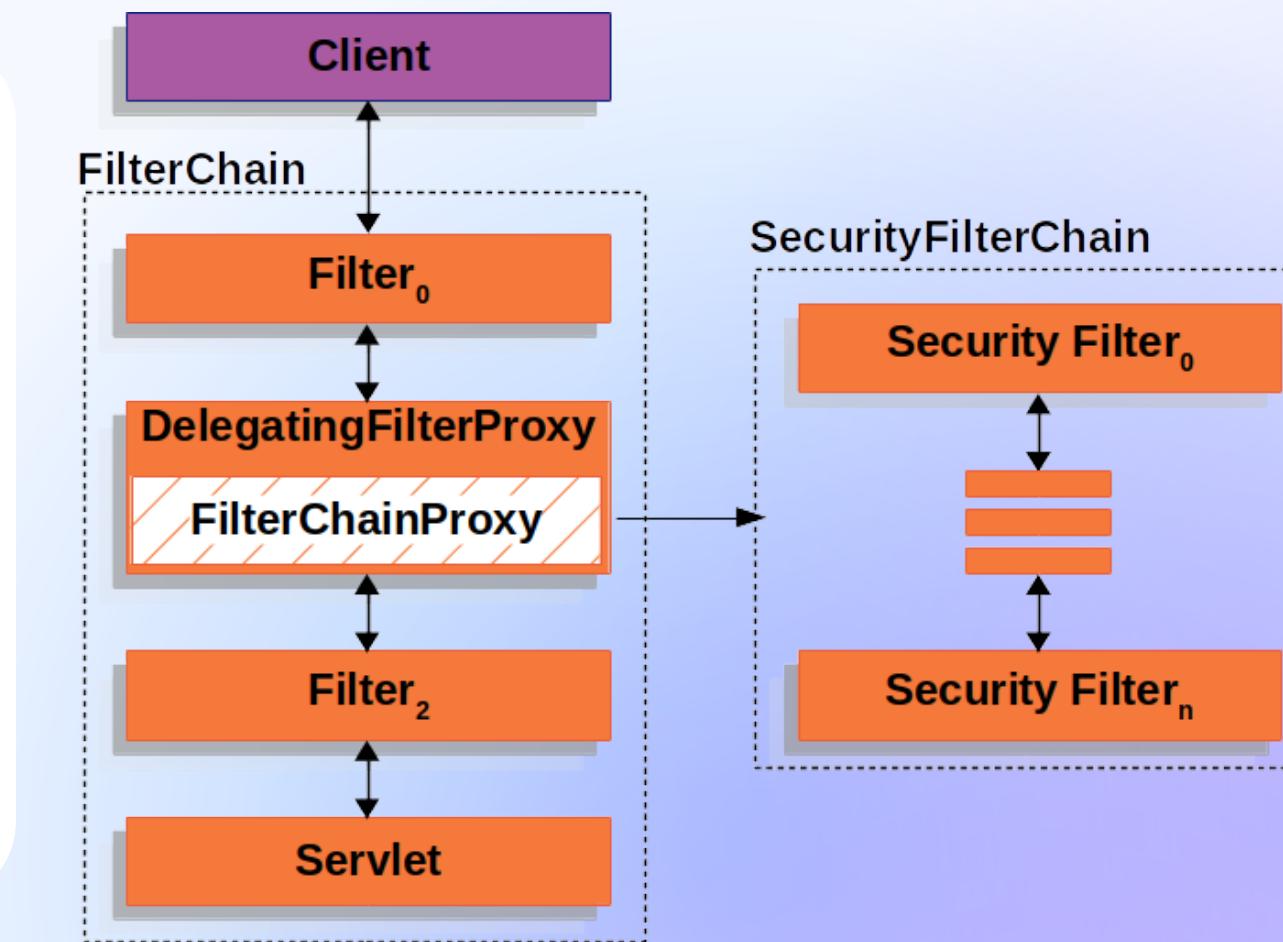
— **FilterChainProxy**



Arquitectura de Spring Security dentro de las aplicaciones basadas en Servlets

Es una interfaz que representa una cadena de filtros de seguridad en Spring Security. Proporciona una forma flexible y poderosa de configurar la seguridad de las solicitudes HTTP en una aplicación web. Cada filtro realiza tareas específicas

— SecurityFilterChain



Arquitectura de autenticación de Spring security parte 1

Es una clase fundamental en Spring Security que proporciona un acceso centralizado al contexto de seguridad en una aplicación. Permite acceder y gestionar la información de autenticación y autorización asociada con una solicitud HTTP en el contexto de seguridad actual.

— **SecurityContextHolder**

Es una interfaz en Spring Security que representa el contexto de seguridad actual de una solicitud. Contiene toda la información relacionada con la autenticación y autorización del usuario que ha realizado la solicitud

— **SecurityContext**



Arquitectura de autenticación de Spring security parte 1

— Principal

Es una representación del usuario autenticado en el sistema. Representa la identidad del usuario actual que está interactuando con la aplicación. El objeto Principal generalmente encapsula los detalles del usuario, como su nombre de usuario, roles, y otra información relacionada con la autenticación

— GrantedAuthority

Es un permiso que es otorgado al principal (usuario logueado): lectura, escritura, eliminacion, actualización sobre un entidad en específico.



Arquitectura de autenticación de Spring security parte 2

Es una interfaz clave en el marco de Spring Security que se utiliza para autenticar a los usuarios en una aplicación. Es el componente central responsable de coordinar y administrar el proceso de autenticación, delegando la autenticación a uno o más AuthenticationProvider configurados en la aplicación

— **AuthenticationManager** y **AuthenticationConfiguration**

Es la implementación más común de AuthenticationManager. Es una clase importante que actúa como un administrador de proveedores de autenticación. Es el componente central responsable de coordinar la autenticación y delegar la responsabilidad de autenticación a los proveedores adecuados.

— **ProviderManager**



Arquitectura de autenticación de Spring security parte 2

Es una interfaz que define el contrato para un proveedor de autenticación. Es una pieza clave del mecanismo de autenticación en Spring Security, ya que permite personalizar cómo se autentica a los usuarios en el sistema

— **AuthenticationProvider**



Arquitectura de autenticación de Spring security parte 2

Es una interfaz que se utiliza para cargar los detalles de un usuario específico, como nombre de usuario, contraseña y roles, desde una fuente de datos externa

— **UserDetailsService**

Es una interfaz que se utiliza para codificar y verificar contraseñas. Su propósito principal es proteger las contraseñas de los usuarios almacenadas en una base de datos u otro sistema de almacenamiento, de manera que no se guarden en texto claro

— **PasswordEncoder**



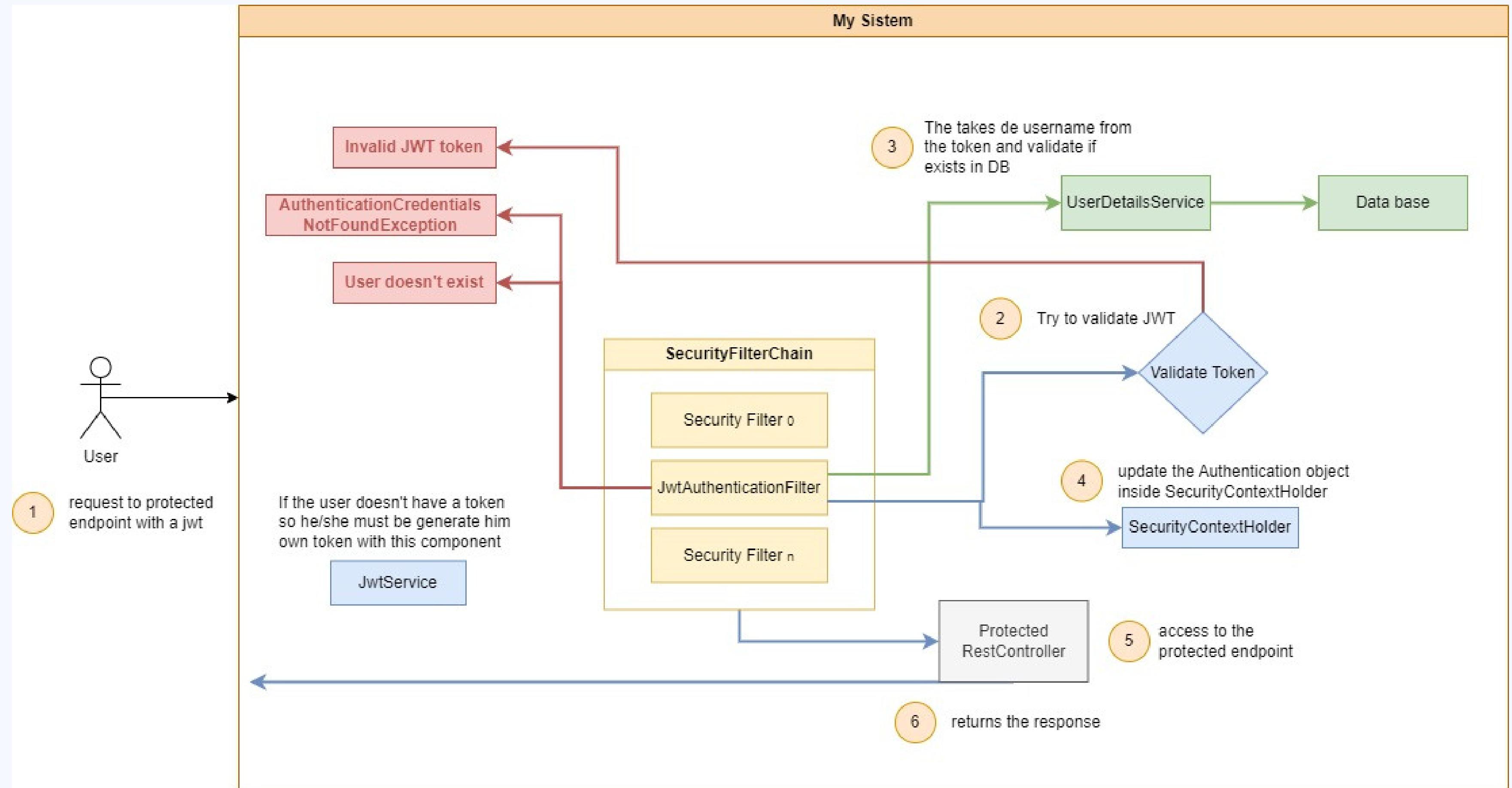


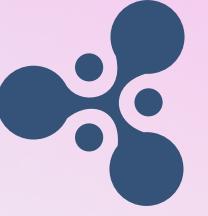
Sección 05:

Arquitectura de

autenticación







Sección 06: Filtros de seguridad.

JwtAuthenticationFilter



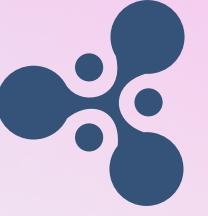


OncePerRequestFilter

OncePerRequestFilter es una clase abstracta proporcionada por Spring Security que simplifica la creación de filtros personalizados para procesar peticiones HTTP una sola vez por solicitud.

La ventaja de utilizar OncePerRequestFilter es que garantiza que el filtro se aplique una sola vez en la cadena de filtros de Spring Security, lo que evita procesamientos innecesarios o duplicados que podrían ocurrir si se usa un filtro común.

Además, al procesar solicitudes en este filtro, se tiene acceso tanto a la solicitud (`HttpServletRequest`) como a la respuesta (`HttpServletResponse`), lo que permite realizar operaciones y tomar decisiones basadas en la información de la solicitud y modificar la respuesta según sea necesario.



Sección 07:

Arquitectura de autorización





Authorities

La autenticación trata sobre cómo todas las implementaciones de autenticación almacenan una lista de objetos `GrantedAuthority`. Estos representan las autoridades/permisos que se han otorgado al principal/username.

"Los objetos `GrantedAuthority` se insertan en el objeto de autenticación."



Invocation Handling

Spring Security proporciona interceptores que controlan el acceso a objetos seguros, como invocaciones de métodos o solicitudes web.

Se toma una decisión antes de la invocación sobre si se permite que la invocación continúe, la cual es realizada por instancias de AuthorizationManager.



AuthorizationManager

Los AuthorizationManager son llamados por los componentes de autorización basados en solicitudes, métodos y mensajes de Spring Security, y son responsables de tomar decisiones finales de control de acceso. La interfaz AuthorizationManager contiene dos métodos:

El método `check` de AuthorizationManager recibe toda la información relevante que necesita para tomar una decisión de autorización. Se espera que las implementaciones devuelvan una `AuthorizationDecision` positiva si se concede el acceso.

El método `verify` llama a `check` y posteriormente lanza una `AccessDeniedException` en caso de una `AuthorizationDecision` negativa.



Implementaciones de AuthorizationManager basadas en Delegados

Si bien los usuarios pueden implementar su propio AuthorizationManager para controlar todos los aspectos de la autorización, Spring Security incluye un AuthorizationManager de delegación que puede colaborar con AuthorizationManagers individuales.

RequestMatcherDelegatingAuthorizationManager coincidirá la solicitud con el AuthorizationManager delegado más apropiado. Para la seguridad de métodos, puedes utilizar AuthorizationManagerBeforeMethodInterceptor y AuthorizationManagerAfterMethodInterceptor.



Tipos de autorizaciones

1. Coincidencia de solicitudes HTTP
2. Asegurar métodos de controladores, servicios, etc. con anotaciones



Tipos de autorizaciones

1 Estrategías de coincidencia de solicitudes HTTP

Spring Security te permite modelar tu autorización a nivel de solicitud. Por ejemplo, con Spring Security puedes indicar que todas las páginas bajo /admin requieren una autoridad, mientras que todas las demás páginas simplemente requieren autenticación.

Por defecto, Spring Security requiere que cada solicitud esté autenticada. Dicho esto, cada vez que uses una instancia de `HttpSecurity`, es necesario declarar tus reglas de autorización.



Tipos de autorizaciones

1 Estrategías de coincidencia de solicitudes HTTP

Spring Security admite dos lenguajes para la coincidencia de patrones de URI: Ant y expresiones regulares, pero hay más:

- Matching Using Ant
- Matching Using Regular Expressions
- Matching By Http Method
- Matching By Dispatcher Type
- Usando un Matcher Customizado

Autorización de peticiones HTTP

Una vez que se coincide con una solicitud gracias a las estrategias mencionadas en la lamina anterior, puedes autorizarla de varias formas.

Como resumen rápido, aquí están las reglas de autorización:

- `permitAll`
- `denyAll`
- `hasAuthority`
- `hasRole`
- `hasAnyAuthority`
- `hasAnyRole`
- `access`

```
authReqConfig
    .requestMatchers(HttpMethod.POST, ...patterns: "/products")
        .hasRole(Role.ROLE_ADMINISTRATOR.name());
authReqConfig.requestMatchers(HttpMethod.POST, ...patterns: "/customers").permitAll();
authReqConfig.requestMatchers(HttpMethod.POST, ...patterns: "/auth/authenticate").permitAll();
authReqConfig.requestMatchers(HttpMethod.GET, ...patterns: "/auth/validate-token").permitAll();

authReqConfig.anyRequest().authenticated();
```



Tipos de autorizaciones

2 Asegurar métodos con anotaciones

El soporte de autorización de métodos de Spring Security es útil para:

1. Extraer lógica de autorización detallada.
2. Aplicar seguridad en la capa de servicios.
3. Estilizar la configuración basada de autorización basada en HttpSecurity (Method Security es mucho más elegante)
4. Y dado que la Seguridad de Métodos se construye utilizando Spring AOP, tienes acceso a todo su poder expresivo.



Tipos de autorizaciones

2 Asegurar métodos con anotaciones

1. @PreAuthorize
2. @PostAuthorize
3. @PreFilter
4. @PostFilter
5. @Secured

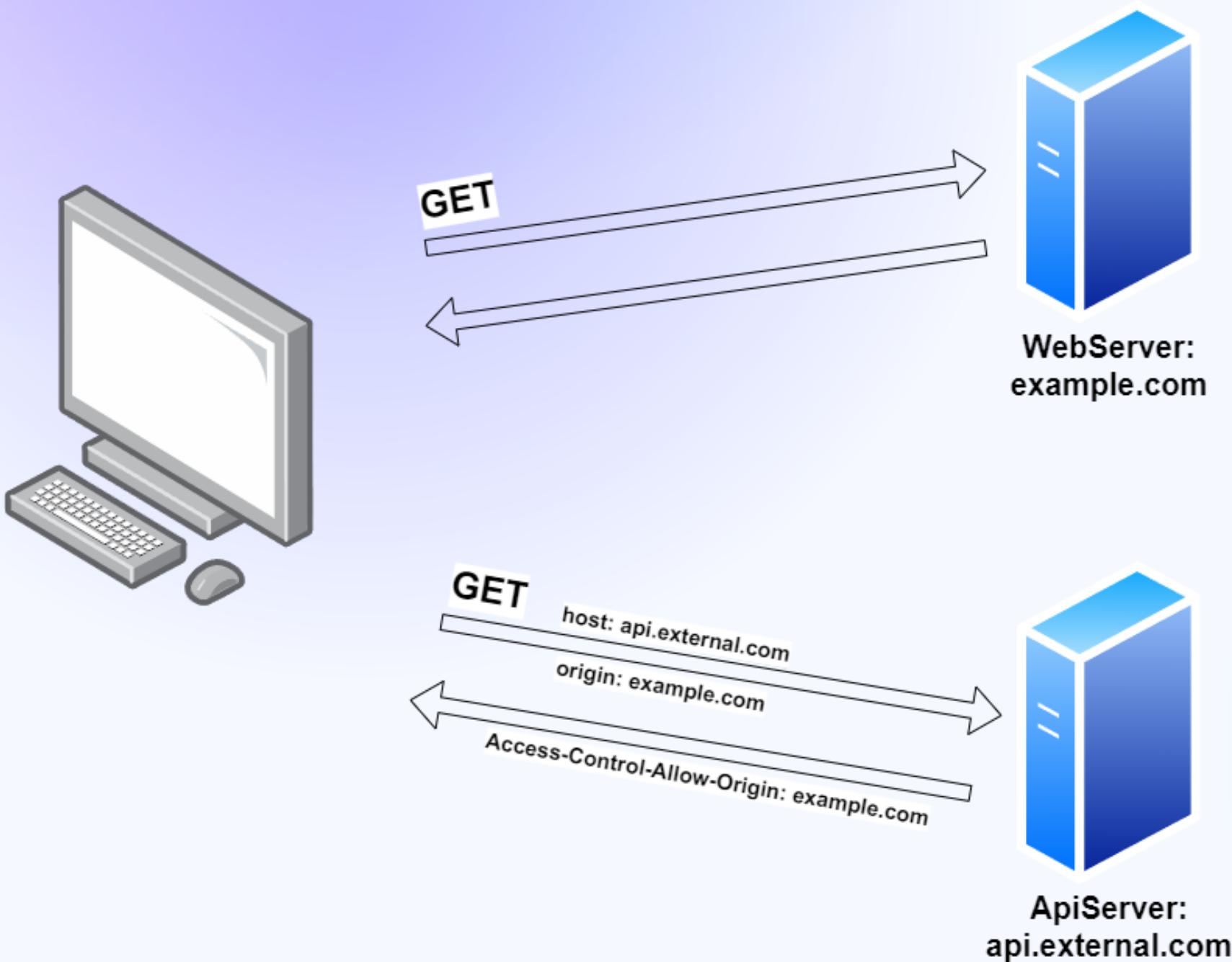
```
@PreAuthorize("hasRole('ROLE_ADMINISTRATOR')")
@PostMapping
public ResponseEntity<Product> createOne(@RequestBody @Valid SaveProduct saveProduct){
    Product product = productService.createOne(saveProduct);
    return ResponseEntity.status(HttpStatus.CREATED).body(product);
}
```



Sección 13: CORS



Cross-Origin Resource Sharing (CORS)



Cross-Origin Resource Sharing (CORS) es un mecanismo de seguridad implementado en los navegadores web para permitir o restringir las solicitudes de recursos entre diferentes dominios (origins) en una aplicación web

CORS aborda la necesidad de permitir excepciones controladas a esta política, permitiendo que los servidores decidan a quién y cómo permitir solicitudes desde otros orígenes.

Encabezados CORS

Los encabezados clave utilizados en el intercambio CORS son:

- **Origin:** Enviado por el navegador e indica el origen del cual proviene la solicitud.
- **Access-Control-Allow-Origin:** Enviado por el servidor e indicar qué orígenes tienen permiso para acceder a los recursos.
- **Access-Control-Allow-Methods:** Indica los métodos HTTP permitidos.
- **Access-Control-Allow-Headers:** Indica las cabeceras que se pueden incluir en la solicitud.
- **Access-Control-Allow-Credentials:** Indica si se permiten credenciales (como cookies) en la solicitud.



Importancia de CORS y Buenas prácticas

- **Configuración del Servidor:** Proporcionar los encabezados **Access-Control-Allow-*** adecuados.
- **Especificidad de Orígenes:** Especifica qué orígenes tienen permisos. No utilices comodines (*).
- **Credenciales:** Asegúrate de configurar **Access-Control-Allow-Credentials** en true para permitir cookies, autenticación HTTP, etc.
- **Validación de Orígenes:** Validar los orígenes permitidos. Los encabezados CORS son fáciles de falsificar.
- **Solicitud Preflight:** Las solicitudes que incluyen ciertas cabeceras personalizadas o utilizan métodos HTTP no seguros (como PUT, DELETE) pueden activar solicitudes preflight. Manejar estas solicitudes enviando respuestas adecuadas a las solicitudes OPTIONS preflight.





Sección 14:

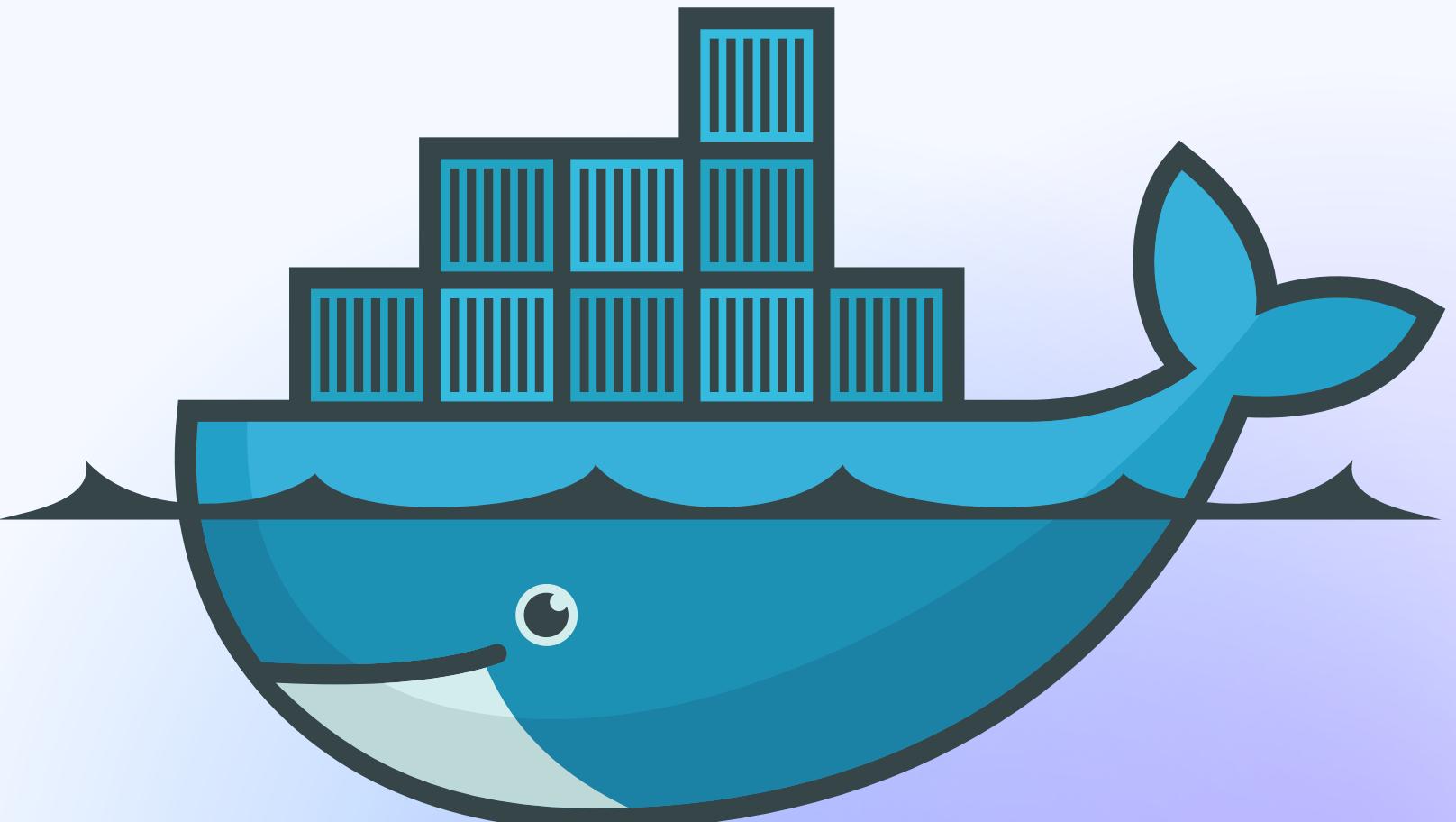
Docker desde

CERO (BONUS)



¿Qué es Docker? Conceptos y beneficios.

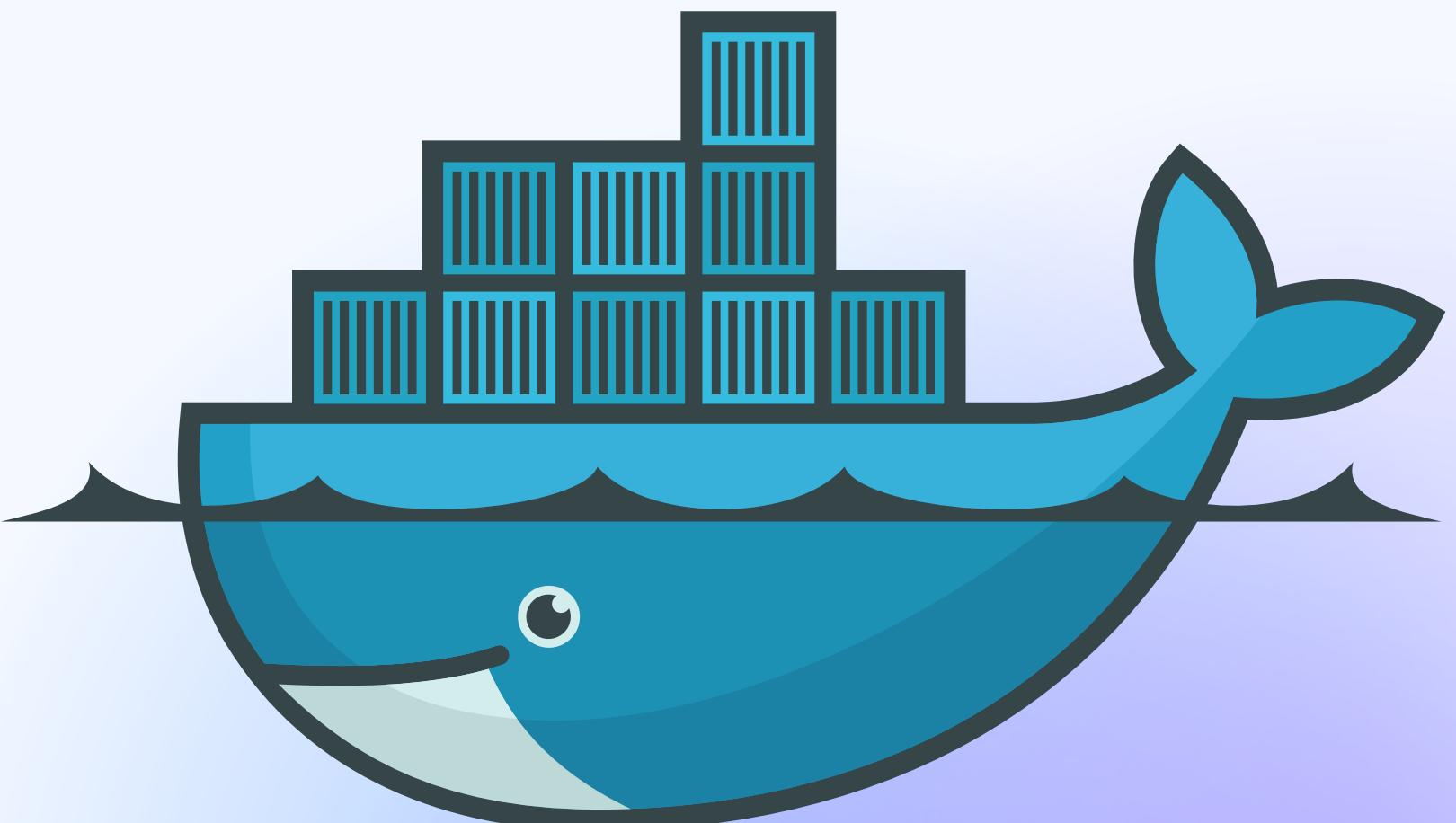
- **Docker es una plataforma de contenedorización que permite empaquetar, distribuir y ejecutar aplicaciones en entornos aislados llamados contenedores.**
- **Estos contenedores incluyen todo lo necesario para que una aplicación funcione, como el código, las bibliotecas, las configuraciones y las dependencias.**
- **Esto garantiza que una aplicación se ejecute de manera coherente en cualquier entorno, ya sea en una máquina local, en servidores de desarrollo o en la nube**



docker

Beneficios clave de Docker

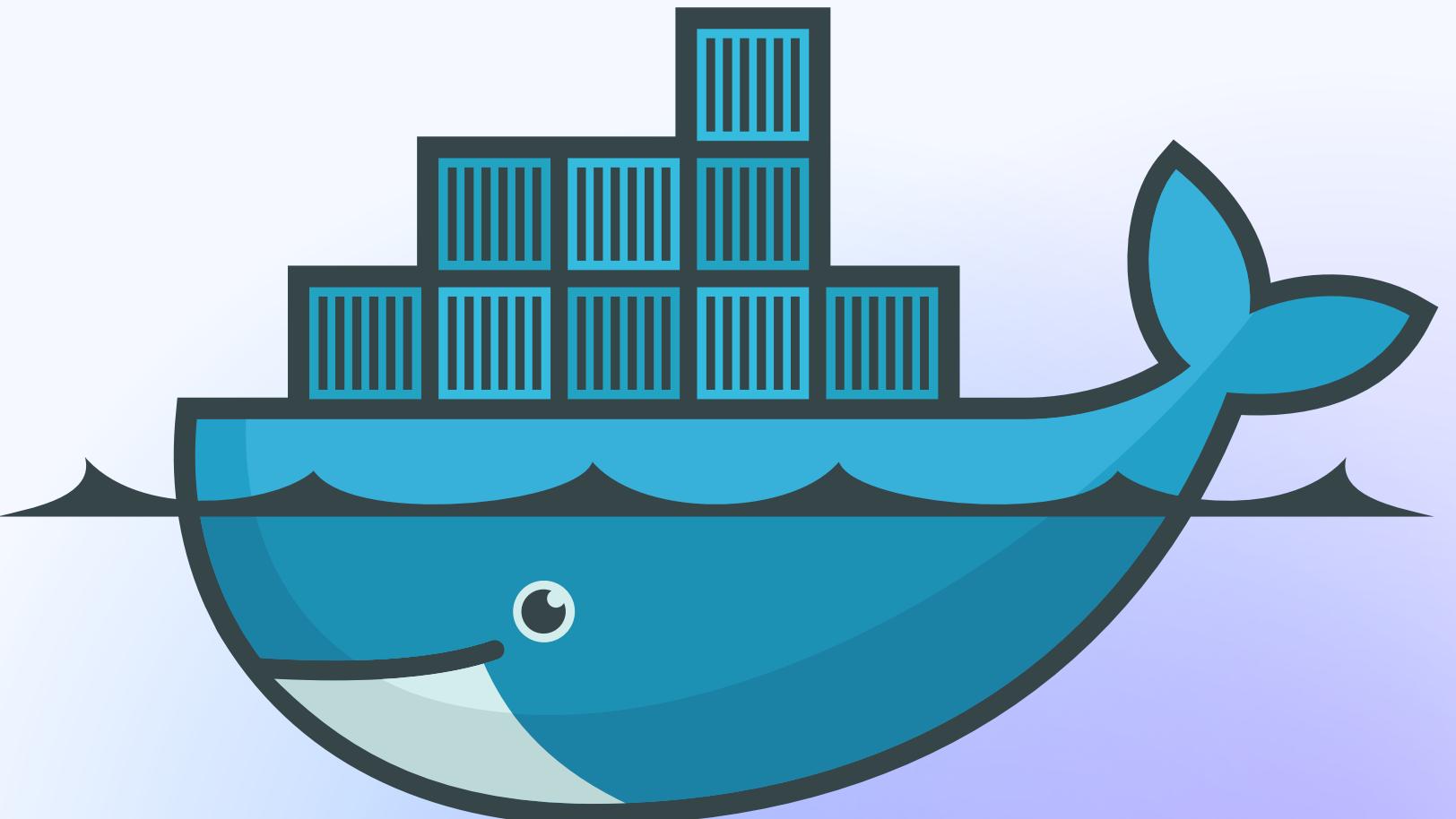
- 👉 **Portabilidad:** empaqueta tus aplicaciones y sus dependencias y ejecutarlas en cualquier entorno sin preocuparse por sistemas operativos.
- 👉 **Aislamiento:** Las aplicaciones no afectarán ni interferirán con otras aplicaciones en el mismo host.
- 👉 **Eficiencia:** Los contenedores comparten el mismo kernel del sistema operativo y aprovechan su eficiencia en comparación con las máquinas virtuales, que requieren sistemas operativos completos.



docker

Beneficios clave de Docker

- 🐳 **Escalabilidad:** Docker facilita la creación y la replicación de contenedores, lo que permite escalar aplicaciones de manera rápida y eficiente según la demanda.
- 🐳 **Desarrollo y despliegue consistentes:** Los entornos de desarrollo, pruebas y producción pueden ser idénticos gracias a la consistencia proporcionada por los contenedores.

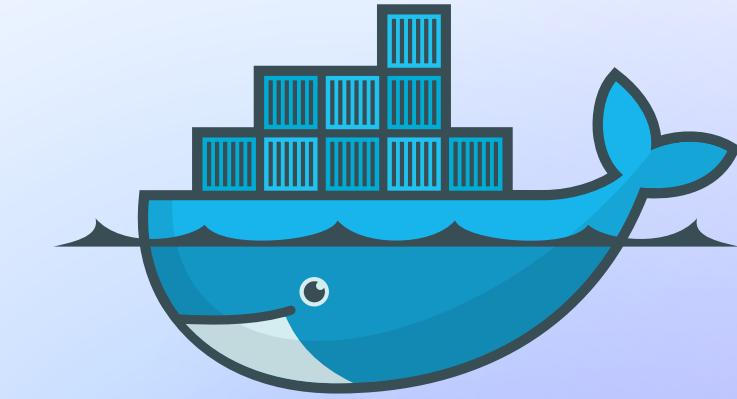


docker

¿Por qué no mejor usar una Máquina Virtual?

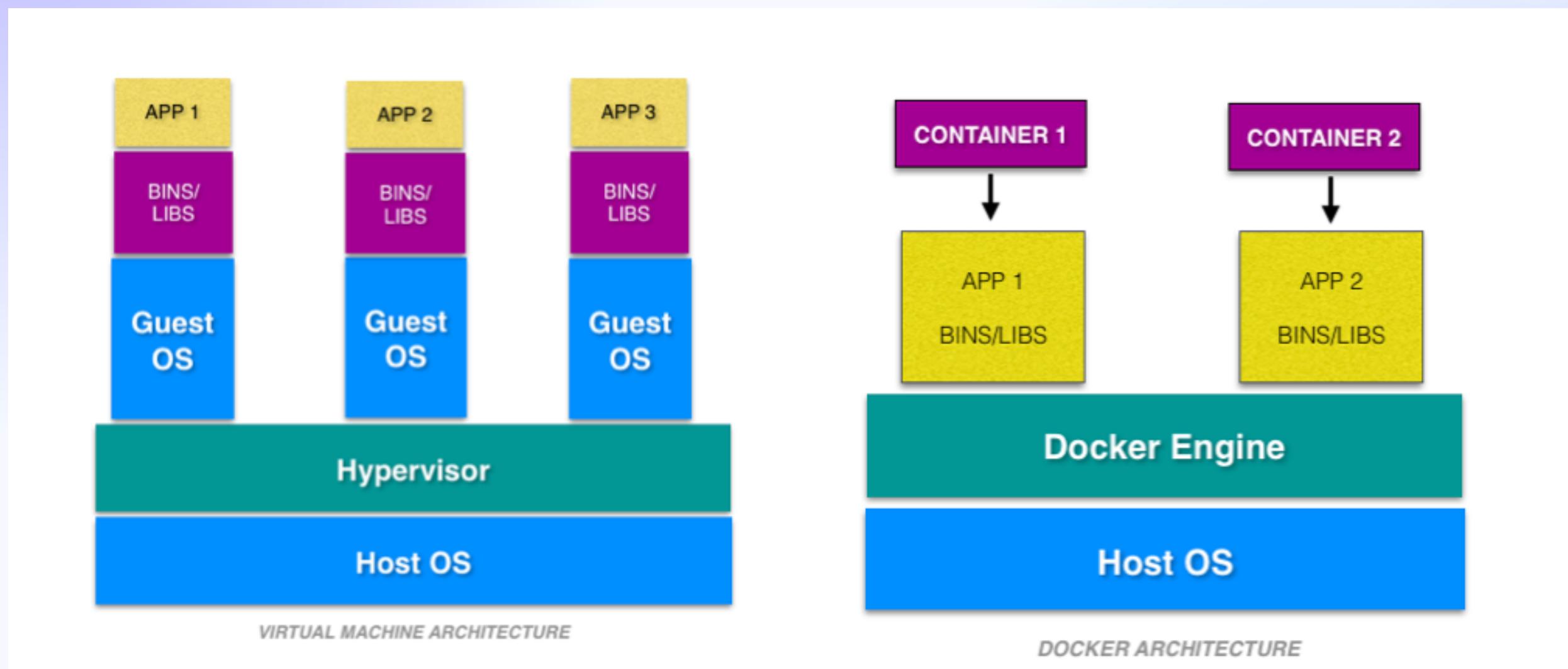
Las máquinas virtuales (VMs) y los contenedores son tecnologías para la virtualización, pero difieren en cómo logran la abstracción y el aislamiento.

Máquinas Virtuales (VMs): Una VM emula un entorno de hardware completo, incluido un sistema operativo, en una capa de software separada. Cada VM requiere su propio sistema operativo y recursos, lo que puede llevar a un uso intensivo de recursos y a la duplicación de sistemas operativos.



Contenedores: Los contenedores se basan en el aislamiento a nivel de sistema operativo. Comparten el mismo kernel del sistema operativo host, lo que resulta en un uso de recursos más eficiente. Los contenedores empaquetan solo lo necesario para ejecutar una aplicación, lo que permite un inicio más rápido y una menor huella de recursos en comparación con las VMs.

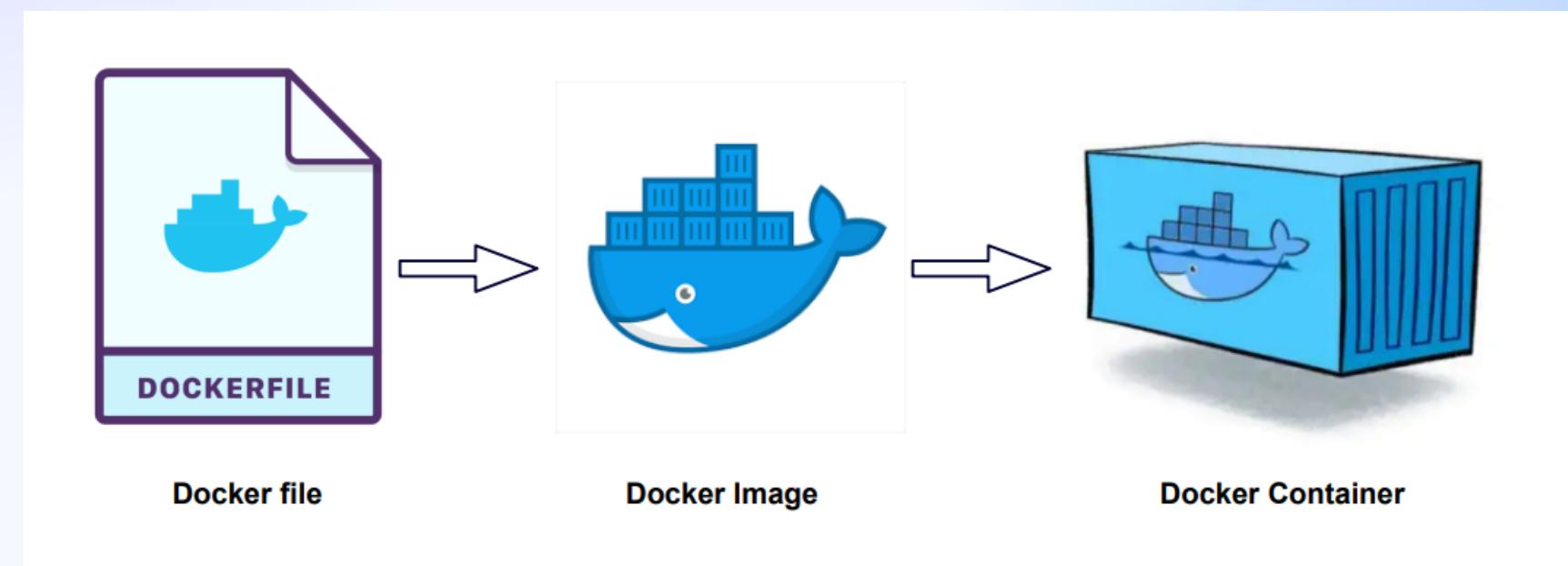
Arquitecturas de las máquinas virtuales vs contenedores



Imágenes y contenedores en Docker.

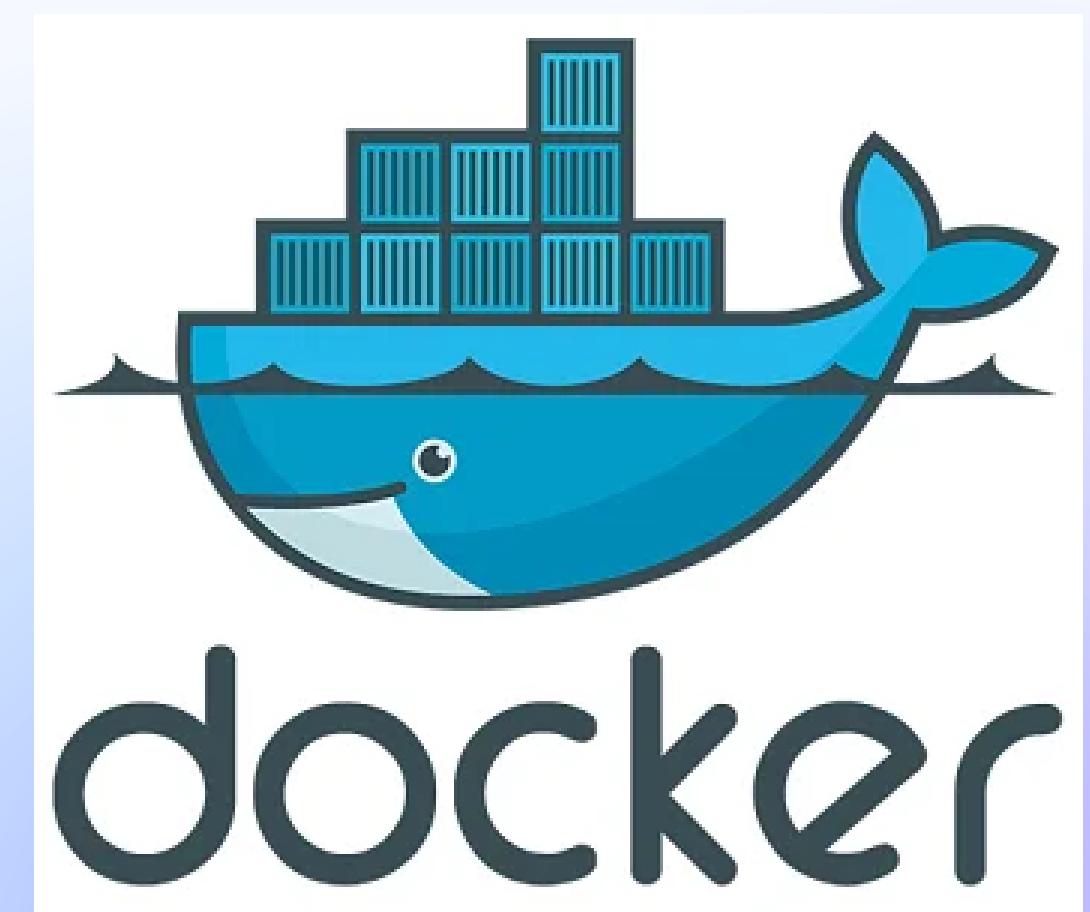
Imágenes: Una imagen de Docker es un paquete ligero y autónomo que contiene todo lo necesario para ejecutar una aplicación, incluyendo el código, las bibliotecas, las dependencias y las configuraciones. Las imágenes se construyen a partir de un archivo llamado Dockerfile, que especifica las instrucciones para crear la imagen.

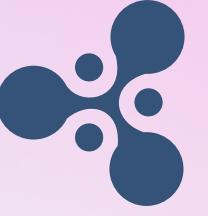
Contenedores: Un contenedor de Docker es una instancia en ejecución de una imagen. Los contenedores son entornos aislados que incluyen su propia copia del sistema operativo, aunque comparten el kernel del sistema operativo del host. Esto permite que las aplicaciones se ejecuten de manera consistente y aislada, sin interferir con otras aplicaciones en el mismo host.



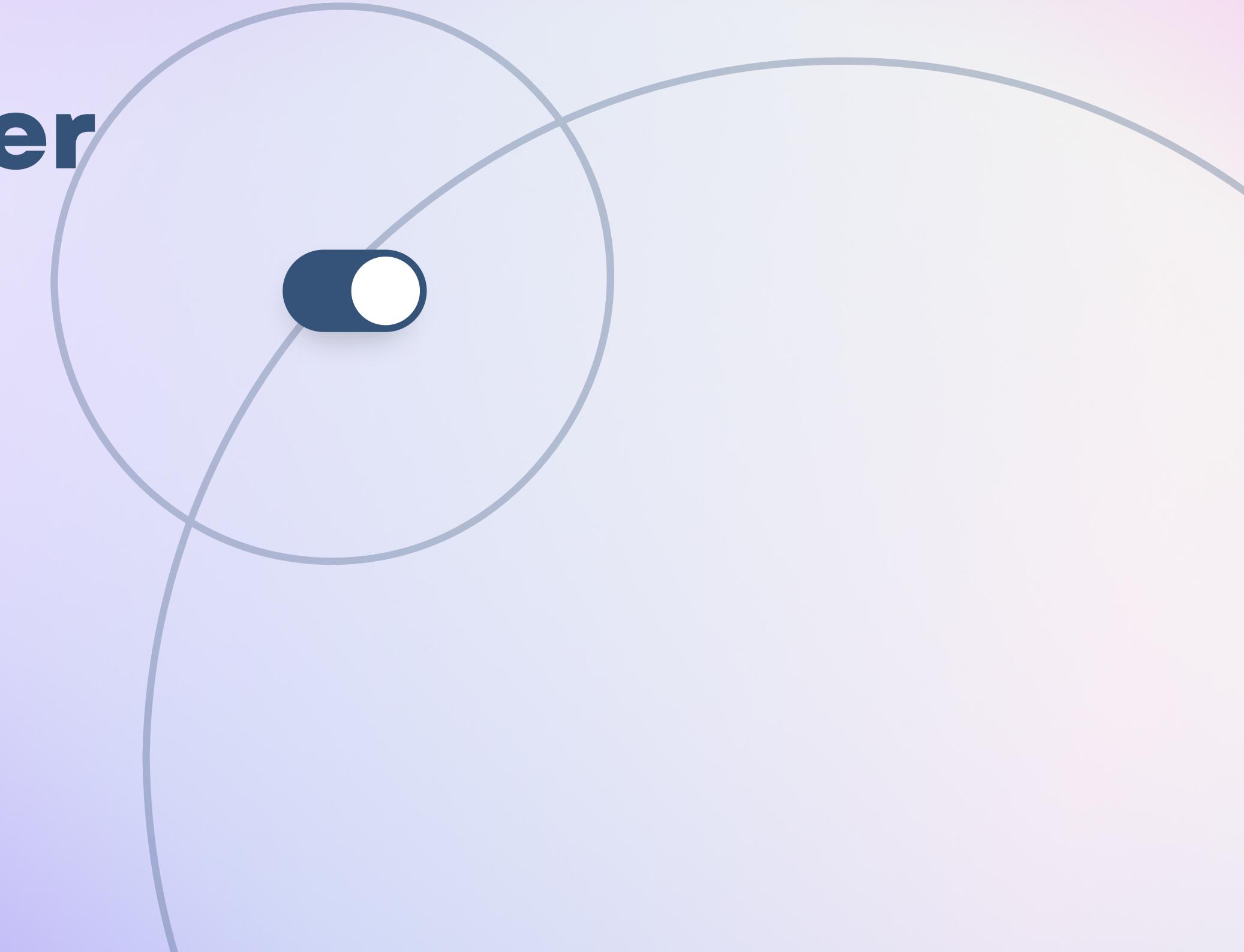
Comandos esenciales de Docker

- `docker pull <nombre de imagen>`: Descarga una imagen desde Docker Hub o un registro personalizado.
- `docker run <nombre de imagen>`: Crea y ejecuta un contenedor a partir de una imagen.
- `docker ps`: Muestra los contenedores en ejecución.
- `docker stop <ID de contenedor>`: Detiene un contenedor en ejecución.
- `docker rm <ID de contenedor>`: Elimina un contenedor.
- `docker images`: Muestra las imágenes disponibles en el registro local.
- `docker rmi <nombre de imagen>`: Elimina una imagen local.
- `docker exec -it <ID de contenedor> <comando>`: Ejecuta un comando en un contenedor en ejecución.
- `docker logs <ID de contenedor>`: Muestra los registros de un contenedor.

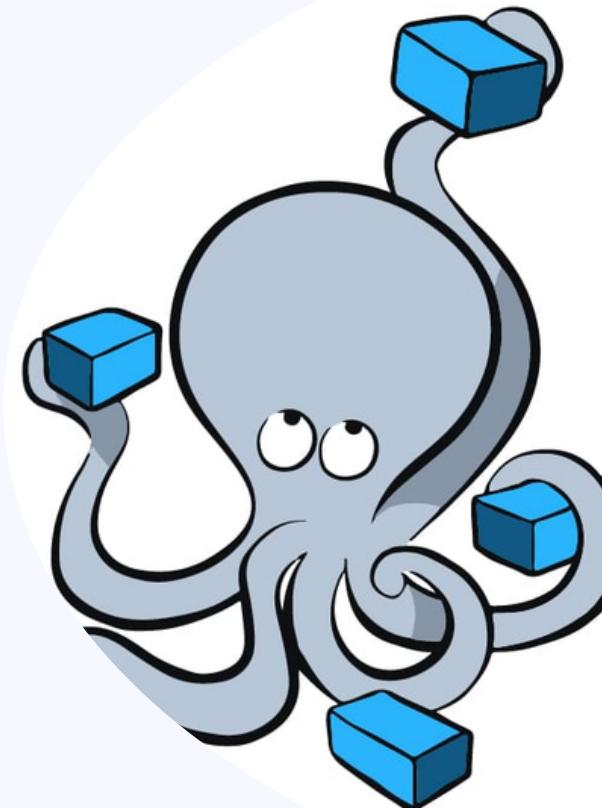




Sección 16: Docker Compose desde CERO(BONUS)



Comandos esenciales de Docker

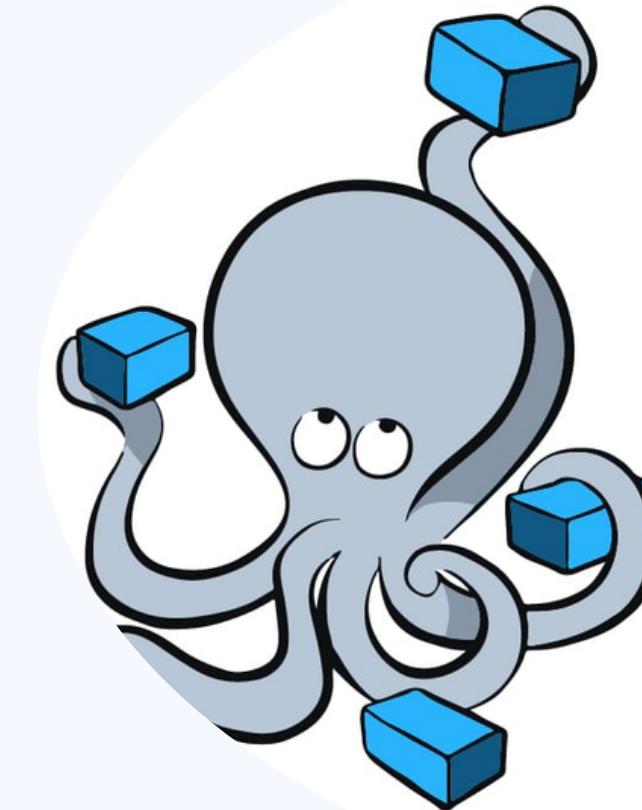


docker
Compose

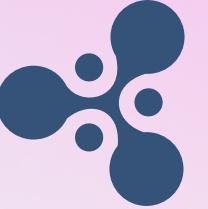
Docker Compose es una herramienta que te permite definir y ejecutar aplicaciones compuestas por múltiples contenedores de Docker como una sola entidad. Es especialmente útil cuando tienes una aplicación que requiere varios servicios, componentes o contenedores que trabajan juntos.

Beneficios de Docker Compose

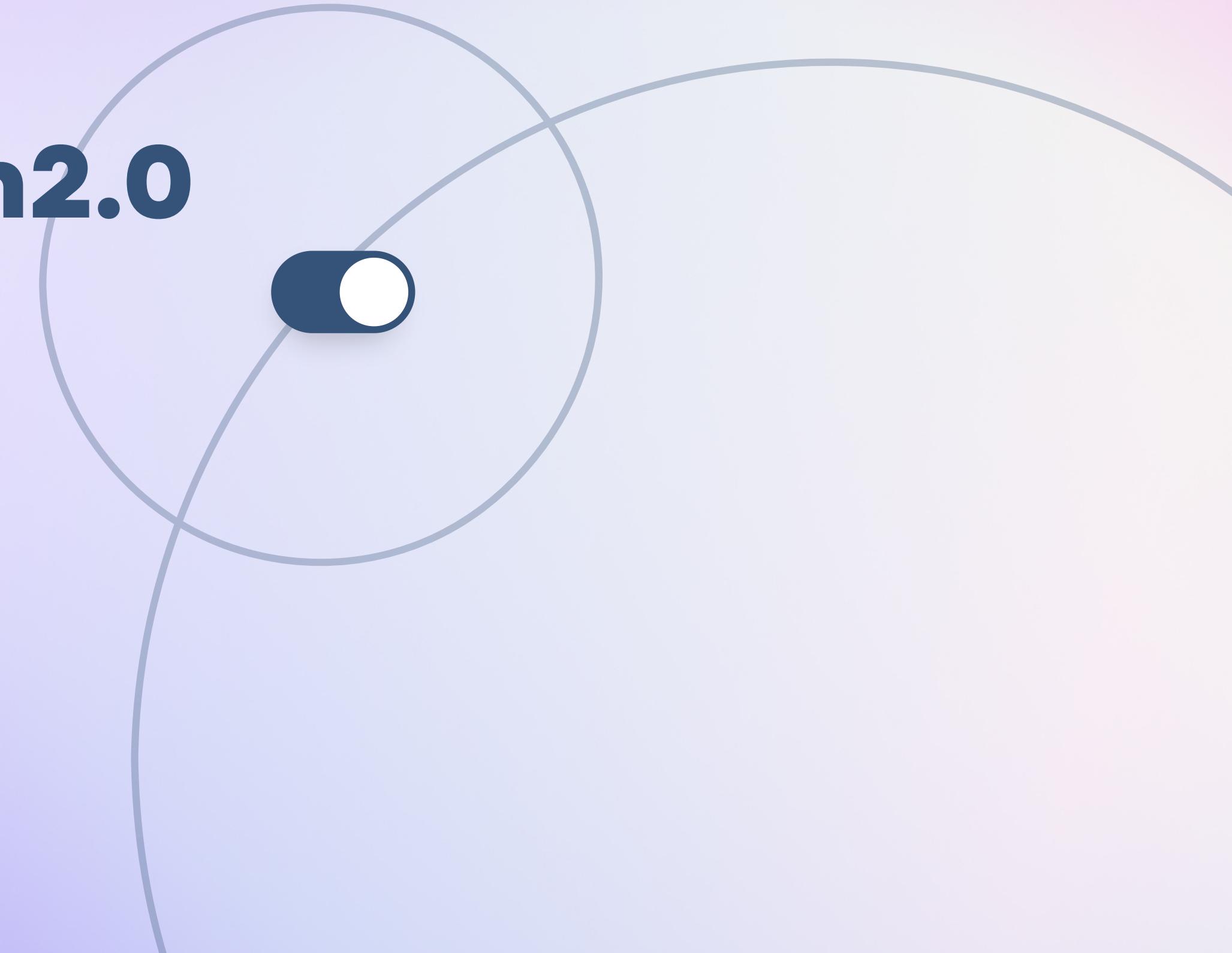
- 1. Orquestación simplificada:** En lugar de tener que ejecutar y vincular manualmente varios contenedores individualmente.
- 2. Configuración en un archivo:** Docker Compose permite definir la configuración de todos los servicios y su relación en un solo archivo docker-compose.yml.
- 3. Ambientes consistentes:** Docker Compose asegura que los contenedores se ejecuten con la misma configuración en todos los entornos (desarrollo, pruebas, producción, etc.).
- 4. Fácil escalabilidad:** Puedes escalar servicios individuales en función de la demanda sin tener que preocuparte por la complejidad de la orquestación manual.
- 5. Aislamiento de servicios:** Aunque los servicios se ejecutan en contenedores separados, Docker Compose permite que estos servicios se comuniquen entre sí de manera sencilla a través de redes internas.
- 6. Depuración y desarrollo más eficientes:** Docker Compose facilita la configuración de entornos de desarrollo replicables.
- 7. Gestión de volúmenes y redes:** Docker Compose permite definir volúmenes y redes personalizadas en el mismo archivo.
- 8. Facilita las pruebas:** Docker Compose facilita la creación de entornos de prueba completos y aislados, lo que mejora la calidad y la confiabilidad del software.



dc
Com



Sección 17: OAuth2.0 desde cero



Introducción a OAuth 2.0

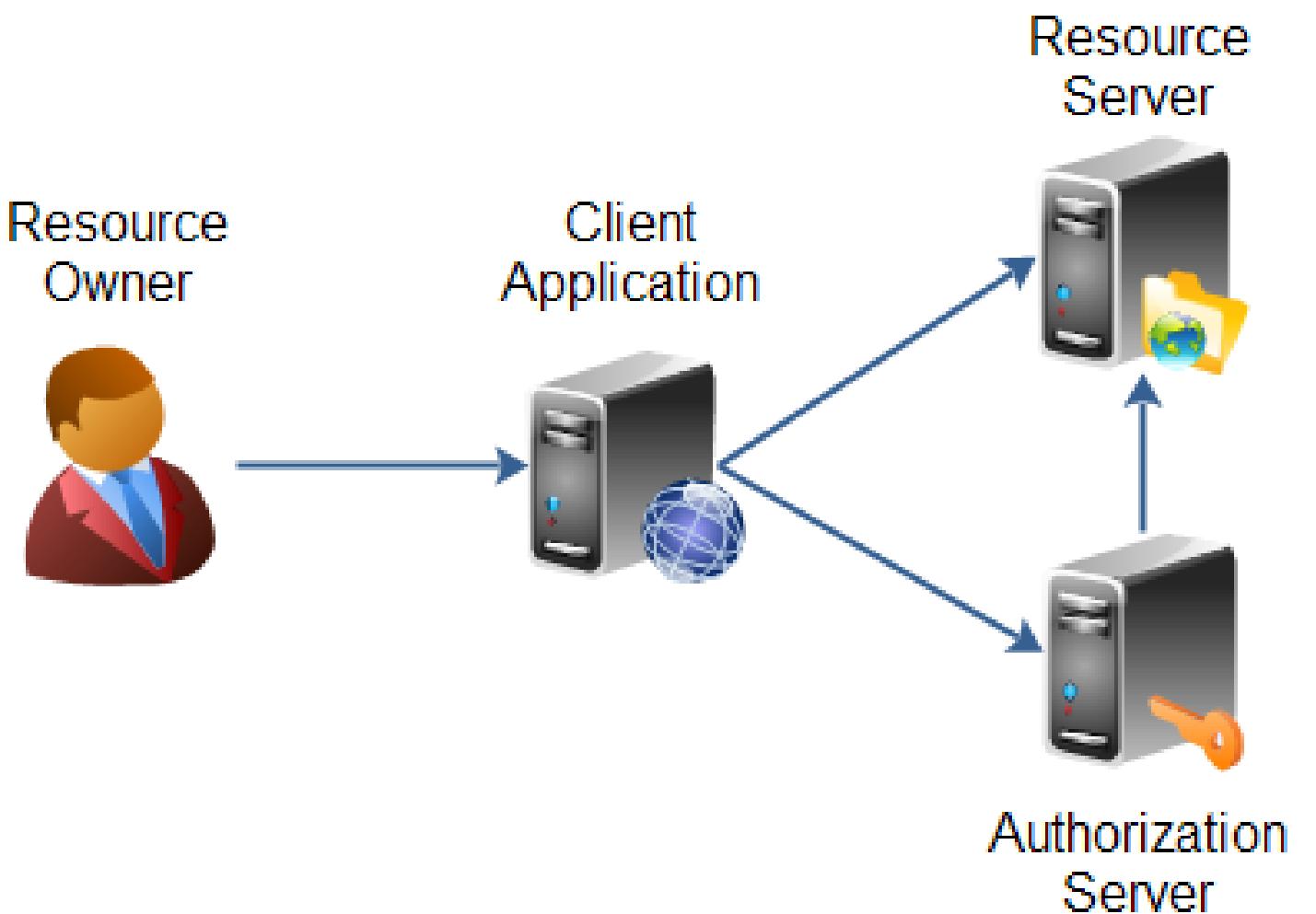
OAuth 2.0, que significa "Open Authorization 2.0", es un protocolo de autorización que se utiliza para permitir que aplicaciones o servicios accedan a recursos protegidos en nombre del propietario de esos recursos, sin que el propietario tenga que compartir sus credenciales directamente con la aplicación. OAuth 2.0 es ampliamente utilizado en la autenticación y autorización de aplicaciones en línea y en la interacción entre servicios web.



Roles en OAuth 2.0

- 1. Cliente (Client):** El cliente es la aplicación o servicio que solicita acceso a los recursos protegidos en nombre del usuario.
- 2. Servidor de Autorización (Authorization Server):** El servidor de autorización es el encargado de autenticar al usuario, obtener su consentimiento y emitir tokens de acceso. Este servidor verifica la identidad del cliente y del usuario, y garantiza que el cliente tenga permiso para acceder a los recursos solicitados.
- 3. Servidor de Recursos (Resource Server):** El servidor de recursos almacena los recursos protegidos que el cliente desea acceder. El servidor de recursos verifica los tokens de acceso y decide si concede o deniega el acceso a los recursos.
- 4. Usuario:** El usuario es el propietario de los recursos protegidos y el que otorga su consentimiento para que el cliente acceda a estos recursos en su nombre.

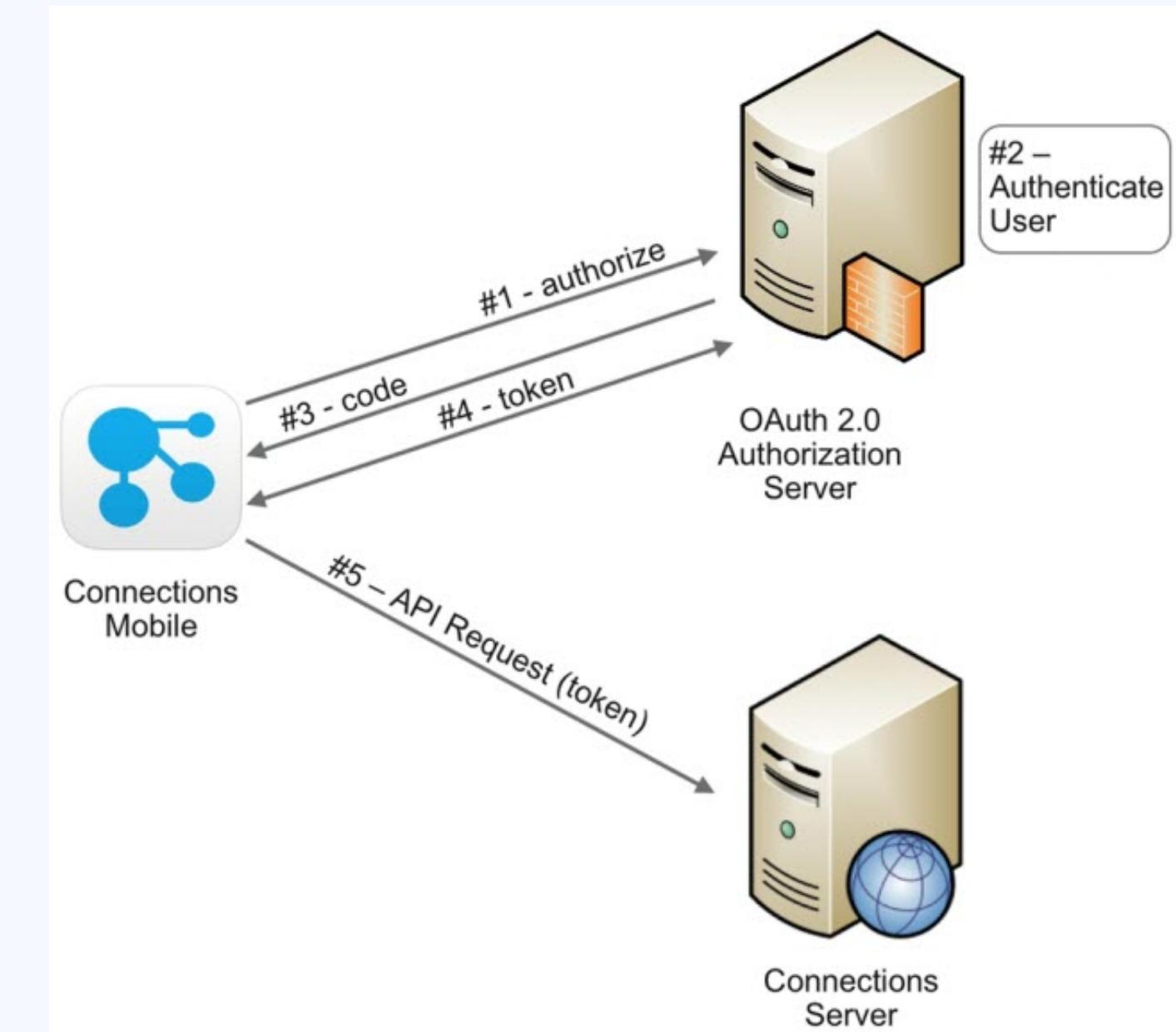
Este rol de "Usuario" es fundamental en OAuth 2.0, ya que representa a la persona que otorga permiso para el acceso a sus datos o recursos protegidos.



Tokens en OAuth 2.0

OAuth 2.0 utiliza varios tipos de tokens para gestionar el acceso a los recursos protegidos. Estos tokens son:

- **Token de Acceso (Access Token):** El Token de Acceso es un código que el servidor de autorización emite al cliente después de que se haya autenticado correctamente y se haya obtenido el consentimiento del usuario. Este token es utilizado por el cliente para acceder a los recursos protegidos en nombre del usuario. Los tokens de acceso son temporales y tienen un tiempo de vida limitado.
- **Refresh Token:** El Refresh Token es un token especial que se utiliza para obtener un nuevo Token de Acceso después de que el anterior haya expirado. Permite que el cliente mantenga el acceso a los recursos protegidos sin requerir que el usuario vuelva a autenticarse y otorgue su consentimiento. Los Refresh Tokens son de larga duración y deben ser almacenados de forma segura.

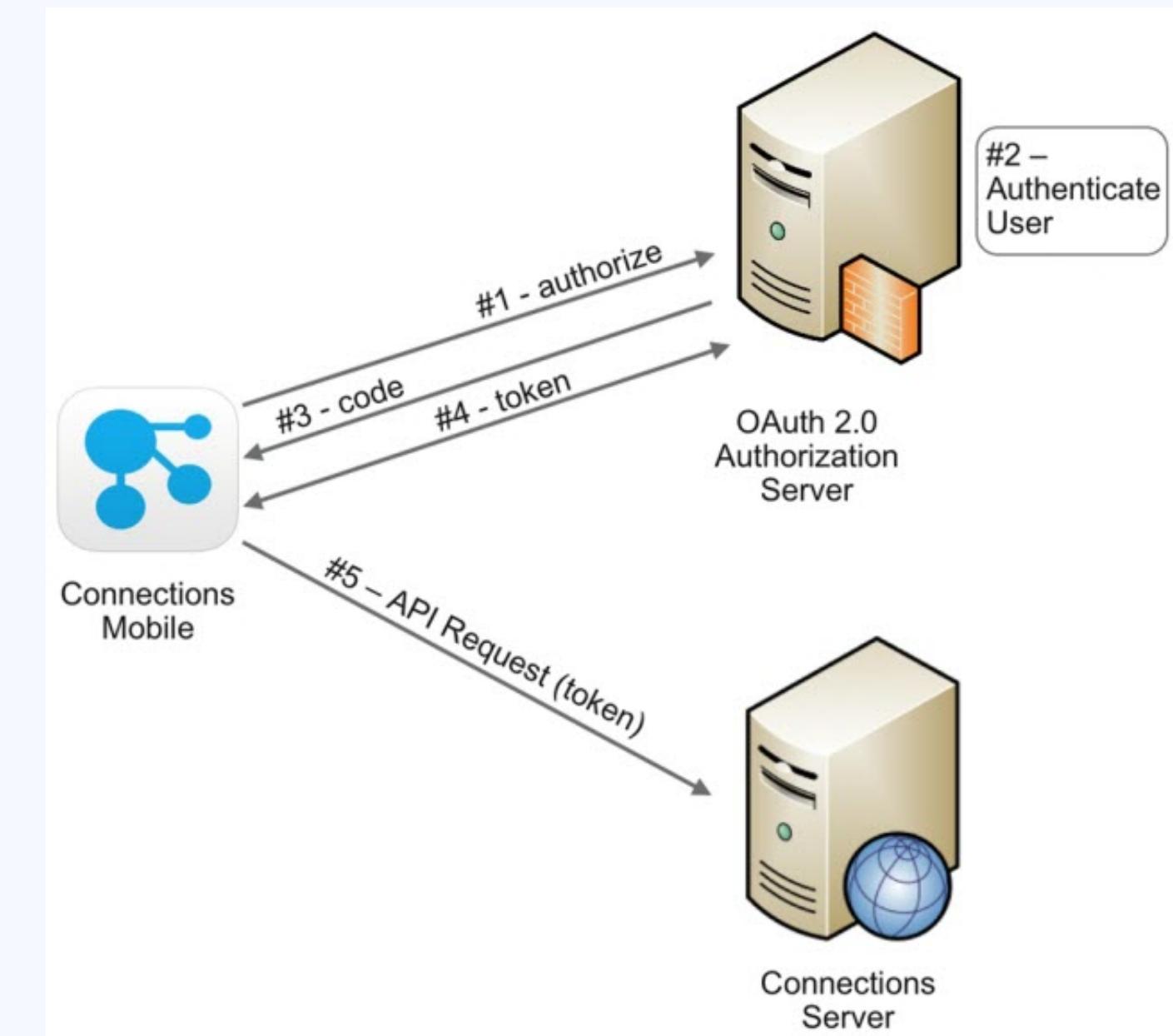


Tokens en OAuth 2.0

OAuth 2.0 utiliza varios tipos de tokens para gestionar el acceso a los recursos protegidos. Estos tokens son:

- **ID Token (Token de Identificación):** Es un token que proporciona información sobre la identidad del usuario. Contiene datos como el nombre del usuario, su dirección de correo electrónico u otra información de perfil.
- **Authorization Code (Código de Autorización):** Aunque técnicamente no es un "token", el Authorization Code es una parte fundamental en el flujo de autorización de OAuth 2.0. Es un código de un solo uso que el cliente obtiene del servidor de autorización y luego intercambia por un Token de Acceso y un Refresh Token. Se utiliza para proteger la transmisión del Token de Acceso durante el proceso de autorización.

Estos tokens son esenciales para la seguridad y la gestión de accesos en OAuth 2.0, y cada uno cumple un propósito específico en el proceso de autorización y autenticación.

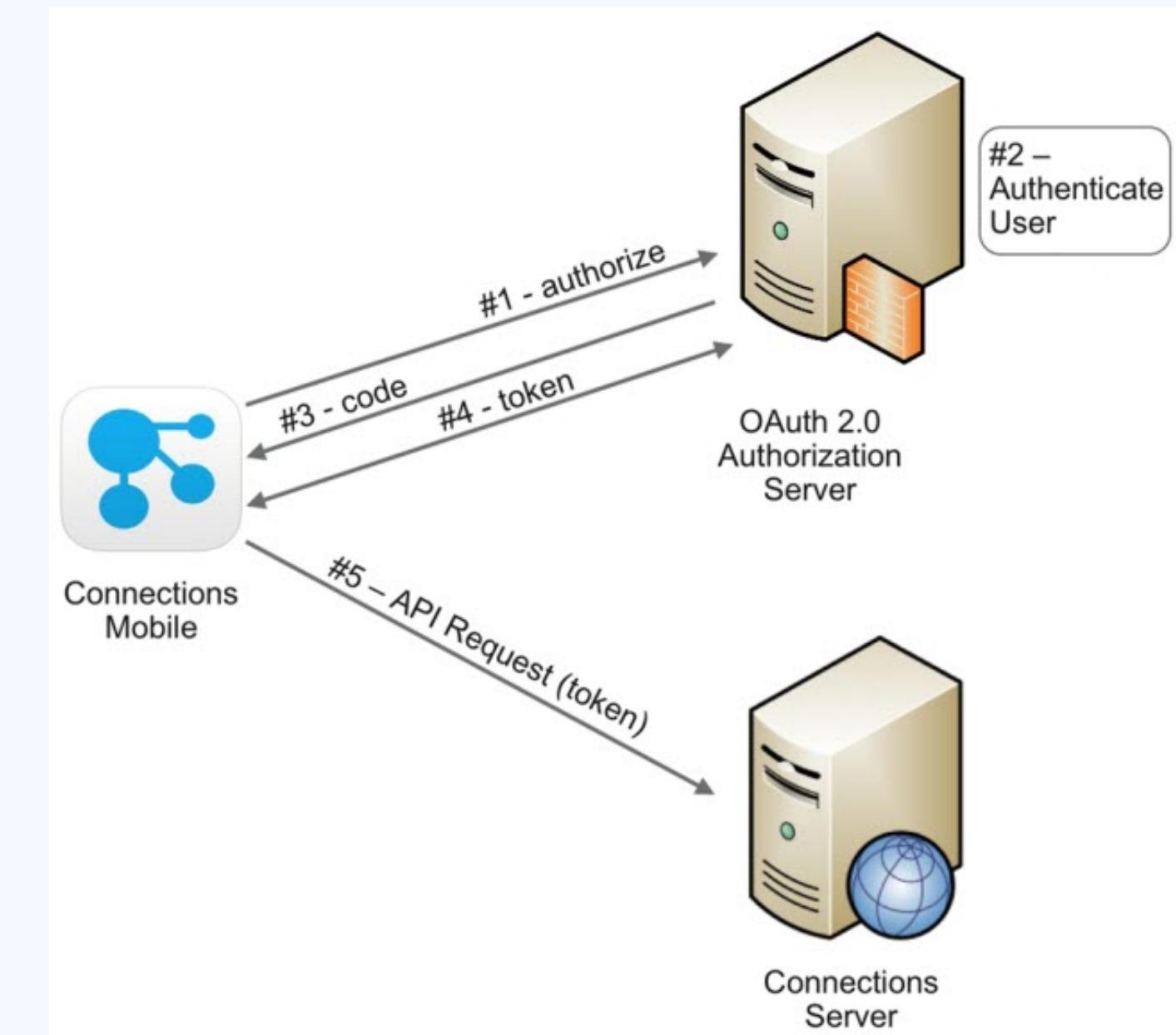


Tokens en OAuth 2.0

OAuth 2.0 utiliza varios tipos de tokens para gestionar el acceso a los recursos protegidos. Estos tokens son:

- **ID Token (Token de Identificación):** Es un token que proporciona información sobre la identidad del usuario. Contiene datos como el nombre del usuario, su dirección de correo electrónico u otra información de perfil.
- **Authorization Code (Código de Autorización):** Aunque técnicamente no es un "token", el Authorization Code es una parte fundamental en el flujo de autorización de OAuth 2.0. Es un código de un solo uso que el cliente obtiene del servidor de autorización y luego intercambia por un Token de Acceso y un Refresh Token. Se utiliza para proteger la transmisión del Token de Acceso durante el proceso de autorización.

Estos tokens son esenciales para la seguridad y la gestión de accesos en OAuth 2.0, y cada uno cumple un propósito específico en el proceso de autorización y autenticación.



Flujos en OAuth2.0

OAuth 2.0 define varios flujos que especifican cómo un cliente puede obtener un token de acceso y cómo se utiliza ese token para acceder a recursos protegidos. Los flujos más comunes incluyen:

- **Authorization Code Flow:** Este flujo es utilizado por aplicaciones web y permite que un cliente obtenga un token de acceso después de que el usuario haya sido redirigido a una página de inicio de sesión y haya otorgado su consentimiento.
- **Authorization Code Flow with PKCE:** Es un flujo diseñado para proporcionar una mayor seguridad cuando una aplicación cliente web se autentica. Ofrece protección adicional contra ataques como la suplantación de código y el robo de tokens.
- **Implicit Flow:** Este flujo es utilizado por aplicaciones de una sola página (SPA) y permite que el cliente obtenga un token de acceso directamente en el navegador del usuario, sin necesidad de un token de autorización (ES MUY POCO RECOMENDADO USAR ESTE FLUJO).