

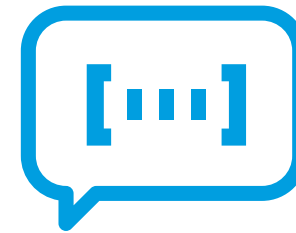
Spring Boot

Seguridad Spring Boot

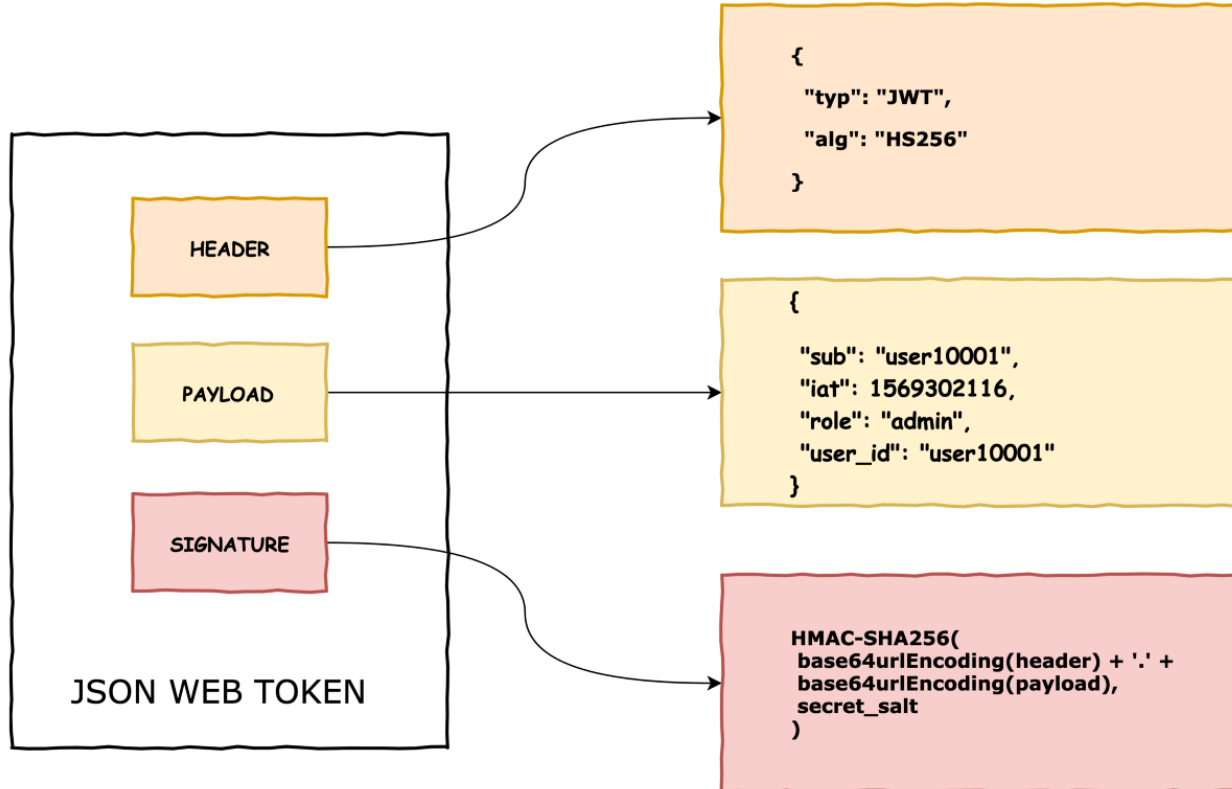


01

JWT

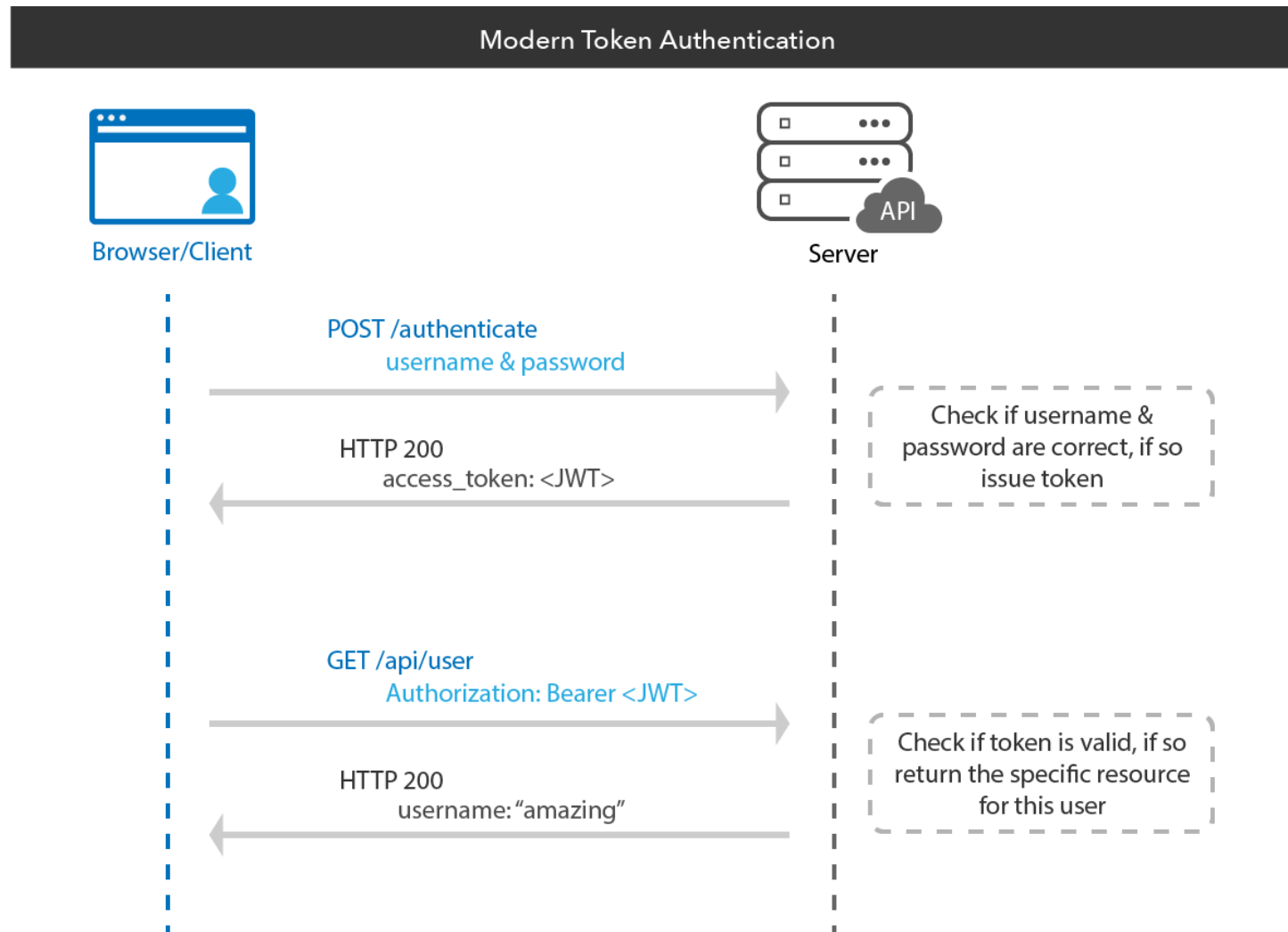


JWT (JSON Web Token)



- JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma de transmitir información de forma segura entre las partes como un objeto JSON.
- Esta información se puede verificar y confiar porque está firmada digitalmente.
- Los JWT se pueden firmar usando un secreto (con el algoritmo HMAC) o un par de claves pública/privada usando RSA o ECDSA.
- <https://jwt.io/>

JWT (JSON Web Token)



Disecccionando el token JWT

Encabezado de token

es un objeto JSON codificado en Base64URL. Contiene información que describe el tipo de token y el algoritmo de firma que se utiliza, como HMAC, SHA256 o RSA.

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

El payload de JWT

contiene algo llamado *claims*, que son declaraciones sobre la entidad (generalmente el usuario) y datos adicionales. Hay tres tipos diferentes de claims: registradas, públicas y privadas. Los claims son la parte más "interesante" de un token JWT, ya que contienen datos sobre el usuario en cuestión.

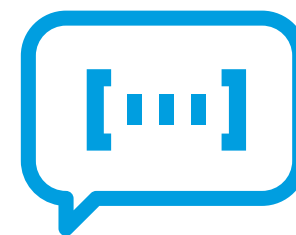
```
{
  "ver": 1,
  "jti": "AT.u_00xGzWwTcDY1xfpp5X_3quR0vRnsnXmwLfWtL1cto",
  "iss": "https://dev-819633.oktapreview.com/oauth2/default",
  "aud": "api://default",
  "iat": 1546726228,
  "exp": 1546729974,
  "cid": "0oaiox8bmsBKVXku30h7",
  "scp": [
    "customScope"
  ],
  "sub": "0oaiox8bmsBKVXku30h7"
}
```

El campo de firma JWT

se crea tomando el encabezado codificado, la carga útil codificada, una clave secreta y usando el algoritmo especificado en el encabezado para firmar criptográficamente estos valores.

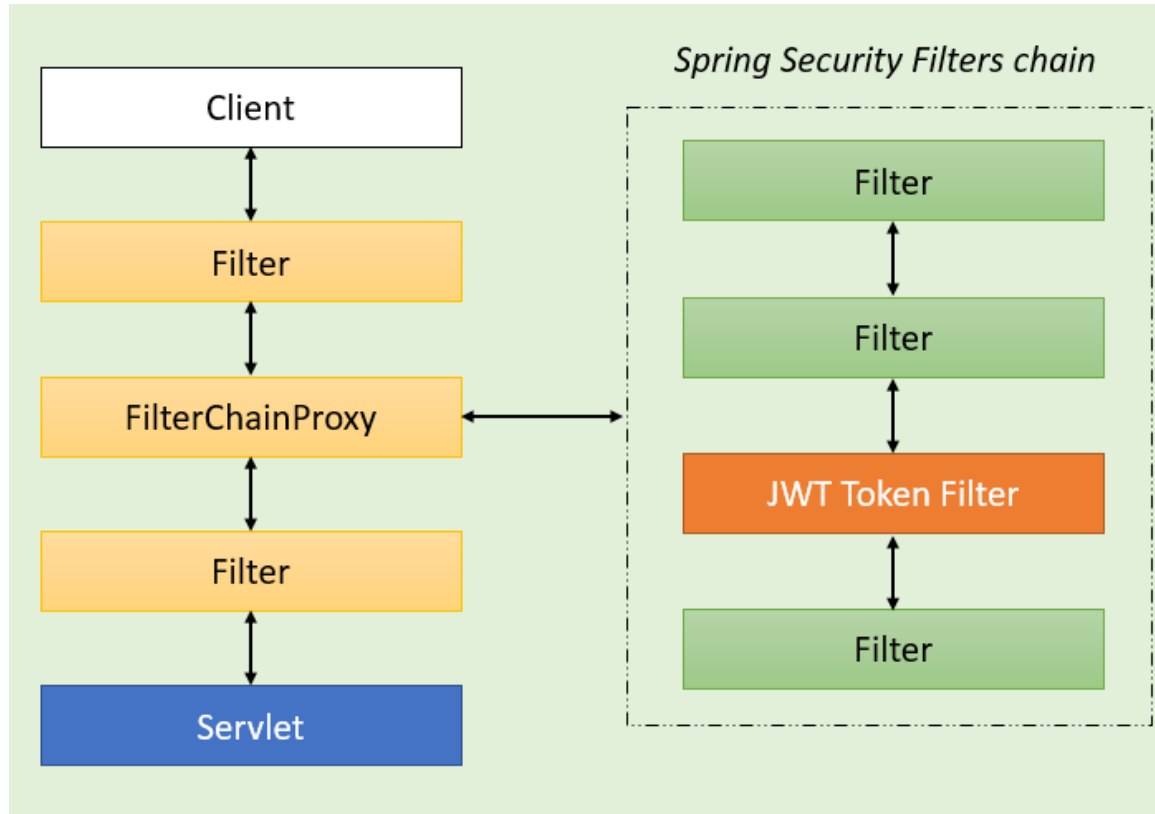
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

02



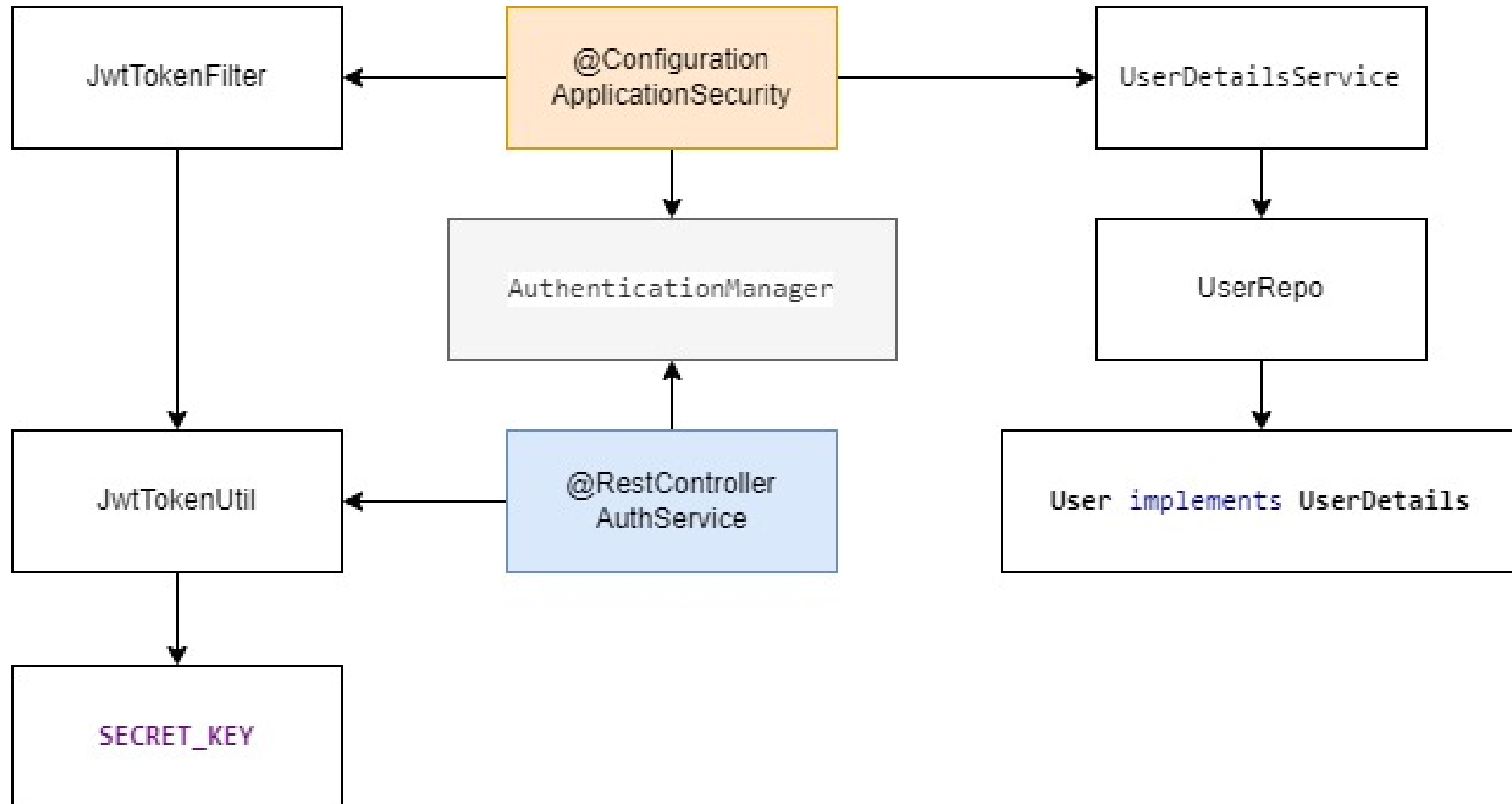
Implementación de JWT con Spring Boot

La cadena de filtros de seguridad Spring



- Cuando un cliente envía una solicitud al servidor, la solicitud pasará por una secuencia de filtros antes de llegar al servlet de destino que es realmente responsable de procesar la solicitud.
- Cada filtro en la cadena de filtros de Spring Security es responsable de aplicar un concern de seguridad específico a la solicitud en curso.

Una implementación simple



Una implementación simple

- **ApplicationSecurity:** es el quid de la implementación de seguridad. Proporciona configuraciones de `HttpSecurity` para configurar cors, csrf, administración de sesiones, reglas para recursos protegidos. Podemos extender y personalizar la configuración por defecto.
- **AuthenticationManager:** tiene un `DaoAuthenticationProvider` (con la ayuda de `UserDetailsService` y `PasswordEncoder`) para validar el objeto token. Si tiene éxito, `AuthenticationManager` devuelve un objeto de autenticación completo (incluidas las autorizaciones otorgadas).
- **JwtTokenFilter:** para acceder a las API REST seguras, el cliente debe incluir un token de acceso en header de Autorización del request. Debemos insertar nuestro propio filtro en el medio de la cadena de filtros de Spring Security, antes de `UsernameAndPasswordAuthenticationFilter`, para verificar el header de Autorización de cada solicitud.
- **JwtTokenUtil:** clase de utilidad que utiliza la biblioteca `jjwt` para generar un token de acceso basado en un objeto de usuario determinado.
- **UserDetailsService:** interfaz que tiene un método para cargar un Usuario por nombre de usuario y devuelve un objeto `UserDetails` que Spring Security puede usar para autenticación y validación.
- **Usuario:** implementa la interfaz `UserDetails` según lo requiere Spring Security. Esta contiene información necesaria, como: nombre de usuario, contraseña, autoridades, para construir un objeto de autenticación.
- **AuthService:** maneja las solicitudes de autenticación.

Dependencias

- Se necesita la siguiente dependencia en el pom.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>
```

La clase de configuración de seguridad

```
@Configuration
public class ApplicationSecurity {

    @Autowired
    private UserRepository userRepo;
    @Autowired
    private JwtTokenFilter jwtTokenFilter;

    @Bean
    public UserDetailsService userDetailsService() {
        return new UserDetailsService() {...};
    }

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

    @Bean
    public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authConfig
    ) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
        ...
    }
}
```

Clase de filtro de token JWT

```
@Component
public class JwtTokenFilter extends OncePerRequestFilter {
    @Autowired
    private JwtTokenUtil jwtUtil;

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

        if (!hasAuthorizationBearer(request)) {
            filterChain.doFilter(request, response);
            return;
        }

        String token = getAccessToken(request);

        if (!jwtUtil.validateAccessToken(token)) {
            filterChain.doFilter(request, response);
            return;
        }

        setAuthenticationContext(token, request);
        filterChain.doFilter(request, response);
    }

    private boolean hasAuthorizationBearer(HttpServletRequest request) {...}

    private String getAccessToken(HttpServletRequest request) {...}

    private void setAuthenticationContext(String token, HttpServletRequest request) {...}

    private UserDetails getUserDetails(String token) {...}
}
```

Clase de utilidad de token JWT

```
@Component
public class JwtTokenUtil {
    private static final long EXPIRE_DURATION = 24 * 60 * 60 * 1000;

    @Value("${app.jwt.secret}")
    private String SECRET_KEY;

    public String generateAccessToken(User user) {
        return Jwts.builder()
            .setSubject(String.format("%s,%s", user.getId(),
user.getEmail()))
            .setIssuer("CodeJava")
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() +
EXPIRE_DURATION))
            .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
            .compact();
    }

    private static final Logger LOGGER =
LoggerFactory.getLogger(JwtTokenUtil.class);

    public boolean validateAccessToken(String token) {...}

    public String getSubject(String token) {...}

    private Claims parseClaims(String token) {...}
}
```

Aquí, el método `generateAccessToken()` crea un token web JSON con los siguientes detalles:

- El **subject** es una combinación del ID y el correo electrónico del usuario, separados por una coma.
- El **emisor** es Netmind
- El token se **emite** en la fecha y hora actual
- El token **debe caducar** después de 24 horas.
- El token **se firma** con una clave secreta, que puede especificar en el archivo `application.properties` o desde la variable de entorno del sistema. Y el algoritmo de firma es HMAC usando SHA-512.

Endpoint de autenticación

```
@RestController
public class AuthService {
    @Autowired
    AuthenticationManager authManager;
    @Autowired
    JwtTokenUtil jwtUtil;

    @PostMapping("/auth/login")
    public ResponseEntity<?> login(@RequestBody @Valid AuthRequest request) {
        try {
            Authentication authentication = authManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                    request.getEmail(), request.getPassword())
            );

            User user = (User) authentication.getPrincipal();
            String accessToken = jwtUtil.generateAccessToken(user);
            AuthResponse response = new AuthResponse(user.getEmail(), accessToken);

            return ResponseEntity.ok().body(response);

        } catch (BadCredentialsException ex) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
}
```

Proceso de autenticación

- Autenticarse contra el endpoint de autenticación

```
curl -v -H "Content-Type: application/json" -d '{"email\":\"my@mail.com\",  
\"password\":\"my_pass\"}' localhost:8080/auth/login
```

- Respuesta

```
{  
  "email": "my@mail.com",  
  "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiIxLG5hbUBjb2RlamF2YS5..."  
}
```

Asegurar la APIs

- Actualicemos el método configure() de la clase de configuración de seguridad.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.authorizeRequests()
        .antMatchers("/auth/login").permitAll()
        .anyRequest().authenticated();
}
```

- Esta configuración expone el punto final /auth/login accesible para todos, pero todas las demás solicitudes deben autenticarse, incluido el URI de /products .

Clase de filtro de token JWT

```
@Component
public class JwtTokenFilter extends OncePerRequestFilter {
    //...continued

    private String getAccessToken(HttpServletRequest request) {
        String header = request.getHeader("Authorization");
        String token = header.split(" ")[1].trim();
        return token;
    }

    private void setAuthenticationContext(String token, HttpServletRequest request) {
        UserDetails userDetails = getUserDetails(token);

        UsernamePasswordAuthenticationToken
            authentication = new UsernamePasswordAuthenticationToken(userDetails, null, null);

        authentication.setDetails(
            new WebAuthenticationDetailsSource().buildDetails(request));

        SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    private UserDetails getUserDetails(String token) {
        User userDetails = new User();
        String[] jwtSubject = jwtUtil.getSubject(token).split(",");

        userDetails.setId(Integer.parseInt(jwtSubject[0]));
        userDetails.setEmail(jwtSubject[1]);

        return userDetails;
    }
}
```

Proceso de solicitud de token y acceso al endpoint

- Autenticarse contra el endpoint de autenticación

```
curl -v -H "Content-Type: application/json" -d "{\"email\":\"my@mail.com\",  
\"password\":\"my_pass\"}" localhost:8080/auth/login
```

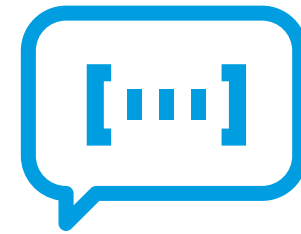
- Respuesta

```
{  
  "email": "my@mail.com",  
  "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiIxLG5hbUBjb2RlamF2YS5..."  
}
```

- Petición a endpoint restringido

```
curl -v -H "Authorization: Bearer <token>" localhost:8080/products
```

03



OAuth2 con JWT

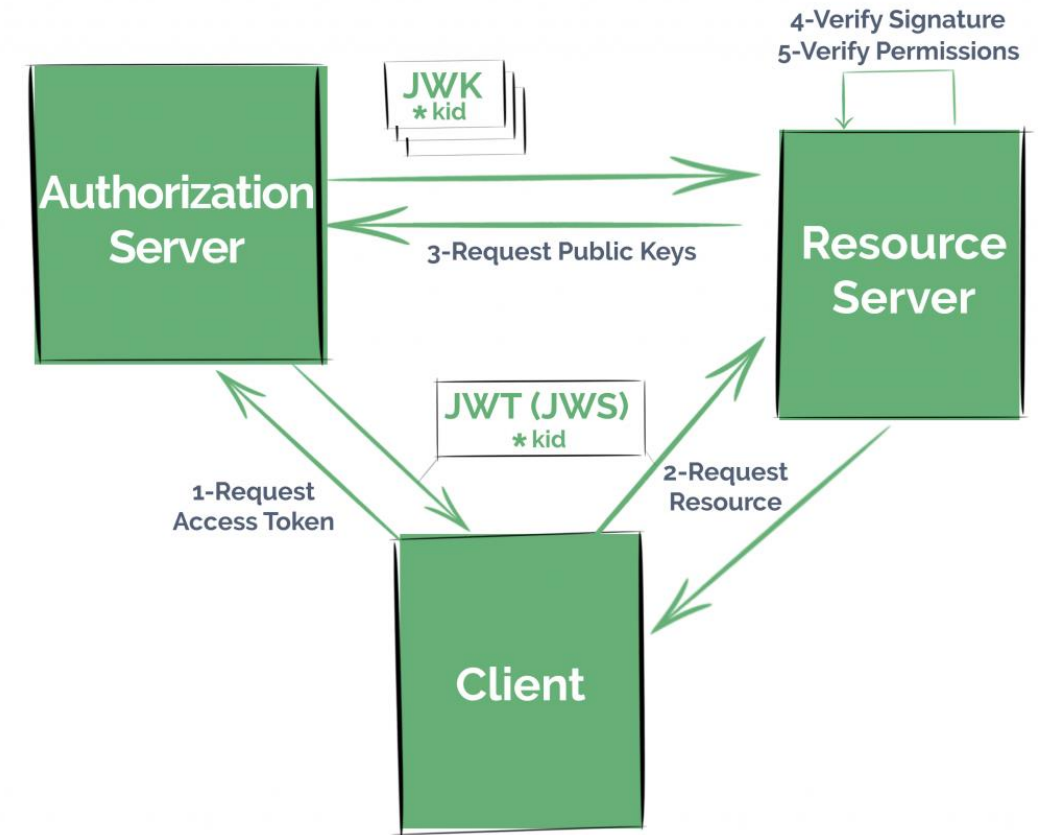
OAuth2



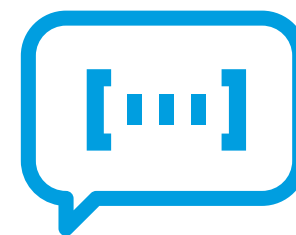
OAuth2 con JWT

¿Cómo funciona?

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



04



Implementación de OAuth2 con JWT en Spring Boot

Spring Security para OAuth2

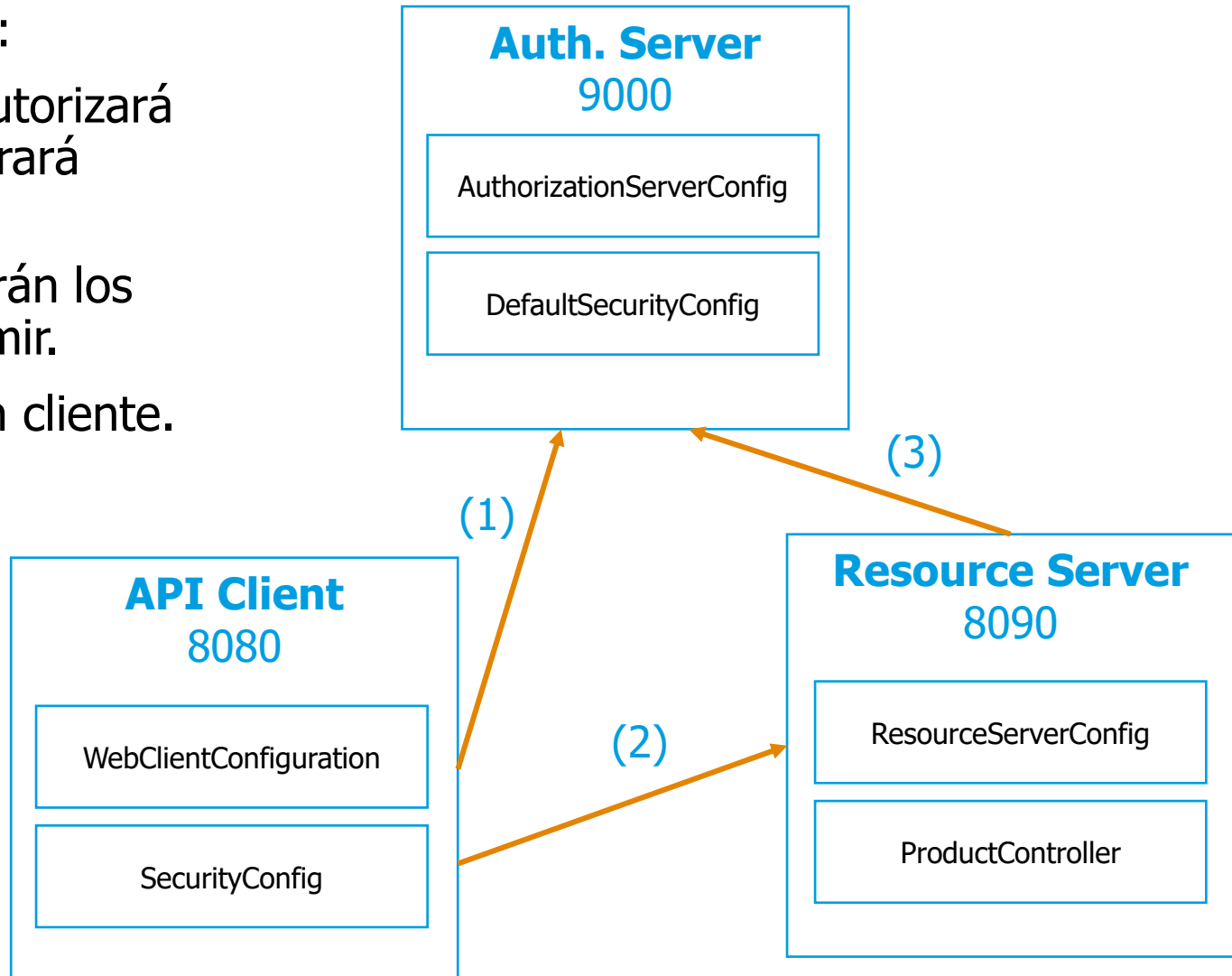
- El antiguo proyecto Spring Security OAuth ha llegado al final de su vida útil (VMware, Inc. ya no lo mantiene activamente).
- Ahora Spring da soporte a OAuth2 con los frameworks:
 - **Spring Security** (<https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html>)
 - **Spring Authorization Server** (<https://spring.io/projects/spring-authorization-server>).
- Spring Authorization Server es un framework que proporciona implementaciones de las especificaciones **OAuth 2.1**, **OpenID Connect 1.0** y otras.
- Está construido sobre Spring Security para proporcionar una base segura, liviana y personalizable.



Arquitectura y configuración

Crearemos 3 proyectos spring-boot:

- 1. Servidor de autorización:** Autorizará el acceso a los recursos y generará tokens de acceso.
- 2. Servidor de recursos:** Exhibirán los recursos que se quieren consumir.
- 3. Cliente API:** Será la aplicación cliente.



1. Servidor de Autorización

- **Dependencias**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-authorization-server</artifactId>
  <version>0.2.2</version>
</dependency>
```

- **application.yml**

```
server:
  port: 9000
```

Configuración de servidor de autorización - Registro de clientes OAuth

- Estas credenciales serán utilizadas por una aplicación cliente cuando se autentique con el servidor de autorización.

```
@Configuration
public class AuthorizationServerConfiguration {

    @Bean
    public RegisteredClientRepository registeredClientRepository() {

        RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
            .clientId("client1")
            .clientSecret("{noop}myClientSecretValue")
            .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
            .redirectUri("http://127.0.0.1:8080/login/oauth2/code/users-client-oidc")
            .redirectUri("http://127.0.0.1:8080/authorized")
            .scope(OidcScopes.OPENID)
            .scope("read")
            //.clientSettings(ClientSettings.builder().requireAuthorizationConsent(true).build())
            .build();

        return new InMemoryRegisteredClientRepository(registeredClient);
    }
}
```

Configuración de servidor de autorización - Registro de clientes OAuth

- **ClientId y ClientSecret** son credenciales que la aplicación OAuth Cliente usará para autenticarse con el servidor. Si el cliente es una aplicación Javascript, no es necesario configurar el valor de Client Secret. Se debe usar PKCE en su lugar.
- **Método de autenticación** del cliente: se establece en `ClientAuthenticationMethod.CLIENT_SECRET_BASIC`. `CLIENT_SECRET_BASIC` es una autenticación básica utiliza ClientID y Client Secret. Se incluirán en la solicitud HTTP concatenados en una sola cadena, en base64. Si los valores se envían en el cuerpo de la solicitud HTTP Post, se usa `CLIENT_SECRET_POST`.
- El **tipo de concesión de autorización** se establece en `AuthorizationGrantType.AUTHORIZATION_CODE`. Se puede usar también `AuthorizationGrantType.REFRESH_TOKEN`. Aparentemente, no todos los tipos de concesión de autorización serán compatibles con el nuevo Spring Authorization Server.
- **redirectUri**: Una vez que el usuario se autentique correctamente en el servidor de autorización, el código de autorización se asociará a `http://127.0.0.1:8080/authorized` (aplicación de cliente).
- **scope**: se puede agregar más de un alcance si es necesario. La lista de otros ámbitos dependerá de la funcionalidad de la aplicación (read, write, profile).

Configuración de servidor de autorización - Beans del servidor OAuth

- Para configurar la seguridad predeterminada

```
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public SecurityFilterChain authorizationServerSecurityFilterChain(HttpSecurity
http) throws Exception {
    OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
    return http.formLogin(Customizer.withDefaults()).build();
}
```

- URL del emisor

```
@Bean
public ProviderSettings providerSettings() {
    return ProviderSettings.builder()
        .issuer("http://auth-server:9000")
        .build();
}
```

Configuración de servidor de autorización - Métodos para firmar el token

```
@Bean
public JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource) {
    return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
}

@Bean
public JWKSource<SecurityContext> jwkSource() throws NoSuchAlgorithmException {
    RSAKey rsaKey = generateRsa();
    JWKSet jwkSet = new JWKSet(rsaKey);

    return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);
}

private static RSAKey generateRsa() throws NoSuchAlgorithmException {
    KeyPair keyPair = generateRsaKey();
    RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
    RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();

    return new RSAKey.Builder(publicKey)
        .privateKey(privateKey)
        .keyID(UUID.randomUUID().toString())
        .build();
}

private static KeyPair generateRsaKey() throws NoSuchAlgorithmException {
    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(2048);

    return keyPairGenerator.generateKeyPair();
}
```

Configuración de seguridad Spring

- Habilitaremos el módulo de seguridad web Spring con una clase de configuración anotada **@EnableWebSecurity**
- **Actualizar los hosts.**
- Añadir:
127.0.0.1 auth-server
en el fichero /etc/hosts

```
@EnableWebSecurity
public class DefaultSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception
    {
        http.authorizeRequests(authorizeRequests ->
            authorizeRequests.anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults());
        return http.build();
    }

    @Bean
    UserDetailsService users() {
        PasswordEncoder encoder =
            PasswordEncoderFactories.createDelegatingPasswordEncoder();

        UserDetails user = User.withUsername("admin")
            .password(encoder.encode("password"))
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Proceso de solicitud de token

1. Identificarse:

`http://127.0.0.1:9000/oauth2/authorize?response_type=code&client_id=<client_id>&redirect_uri=<redirect_url>/authorized&scope=openid <scope>`

- **Ej:** `http://127.0.0.1:9000/oauth2/authorize?response_type=code&client_id=client1&redirect_uri=http://127.0.0.1:8080/authorized&scope=openid SCOPE_products.read`
- Se recibe una respuesta como: `http://127.0.0.1:8080/authorized?code=<CODE>`

2. Solicitar el token:

En postman lanzar la siguiente request:

- POST `http://127.0.0.1:9000/oauth2/token`
- Authorization (pestaña):
 - Username: `<client_id>`
 - Password: `<client_secret>`
- Body (pestaña):
 - `grant_type: authorization_code`
 - `code: <CODE>`
 - `redirect_uri: <redirect_uri>`
- El valor **access_token** de la respuesta es el que necesitamos para consumir recursos.

Usando Curl:

```
curl -X POST -H "Content-Type: application/x-www-form-urlencoded" -d  
"grant_type=authorization_code&code=<CODE>&redirect_uri=<redirect_uri>&client_id=<client_id>&client_secret=<client_secret>" http://127.0.0.1:9000/oauth2/token
```

2. Servidor de Recursos

- **Dependencias**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
  <version>0.2.2</version>
</dependency>
```

- **application.yml:** Necesitamos configurar la URL del servidor de autenticación

```
server:
  port: 8090

spring.security.oauth2.resourceserver.jwt.issuer-uri: http://auth-server:9000
```


Configuración de seguridad Spring

- Queremos indicar explícitamente que cada solicitud de productos debe estar autorizada y tener la autoridad de lectura de productos (SCOPE_products.read).

```
@EnableWebSecurity
public class ResourceServerConfig {
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .mvcMatchers("/products/**").authenticated()
            .mvcMatchers("/products/**").hasAuthority("SCOPE_products.read")
            .and()
            .oauth2ResourceServer()
            .jwt();
        return http.build();
    }
}
```

Proceso de solicitud de recursos

1. Identificarse y obtener un token contra el servidor de autorización.
 - Copiar el valor **access_token** de la respuesta.
2. Hacer una solicitud al endpoint del recurso del tipo Authorization Bearer añadiendo el token
 - Por ejemplo:
 - GET localhost:8090/products
 - Authorization Bearer: <**access_token**>

Usando Curl:

```
curl -H "Accept: application/json" -H "Authorization: Bearer <access_token>" localhost:8090/products
```

3. Cliente API

Dependencias

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>

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

server:
  port: 8090

spring.security.oauth2.resourceserver.jwt.issuer-uri: http://auth-server:9000

</dependency>
```

Cliente API

application.yml: Definimos las propiedades de configuración para fines de autenticación.

```
server:
  port: 8080

spring:
  security:
    oauth2:
      client:
        registration:
          products-client-oidc:
            provider: spring
            client-id: client1
            client-secret: myClientSecretValue
            authorization-grant-type: authorization_code
            redirect-uri: "http://127.0.0.1:8080/login/oauth2/code/{registrationId}"
            scope: openid
            client-name: products-client-oidc
          products-client-authorization-code:
            provider: spring
            client-id: client1
            client-secret: myClientSecretValue
            authorization-grant-type: authorization_code
            redirect-uri: "http://127.0.0.1:8080/authorized"
            scope: SCOPE_products.read
            client-name: products-client-authorization-code
        provider:
          spring:
            issuer-uri: http://auth-server:9000
```

Configuración del Cliente

- Creamos una instancia de WebClient para realizar solicitudes HTTP al servidor de recursos. Usaremos la implementación estándar con filtro de autorización de OAuth:

```
@Configuration
public class WebClientConfiguration {
    @Bean
    WebClient webClient(OAuth2AuthorizedClientManager authorizedClientManager) {
        ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
            new ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager);
        return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build();
    }
    ...
}
```

Configuración del Cliente

- El WebClient requiere un OAuth2AuthorizedClientManager como dependencia.

```
@Configuration
public class WebClientConfiguration {
    ...
    @Bean
    OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository
    ) {

        OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                .authorizationCode()
                .refreshToken()
                .build();

        DefaultOAuth2AuthorizedClientManager authorizedClientManager = new DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository);
        authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

        return authorizedClientManager;
    }
}
```

Configuración de seguridad Spring

- Configuramos la seguridad web con el cliente adecuado.

```
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests.anyRequest().authenticated()
            )
            .oauth2Login(oauth2Login ->
                oauth2Login.loginPage("/oauth2/authorization/products-client-oidc"))
            .oauth2Client(Customizer.withDefaults());
        return http.build();
    }
}
```

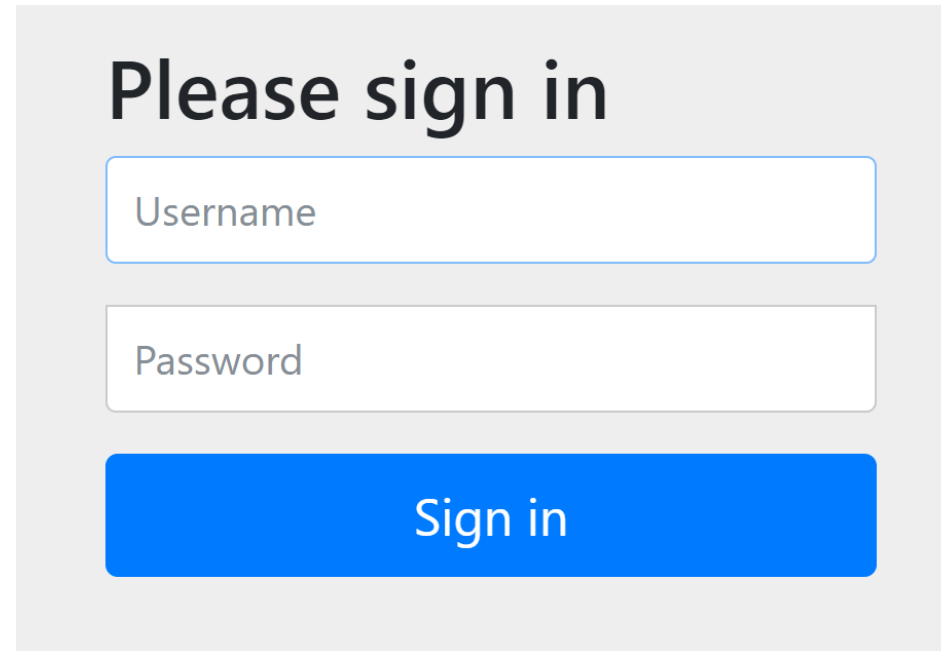
La vista del cliente

- Finalmente, podemos crear el controlador de acceso a datos. Usaremos el WebClient previamente configurado para enviar una solicitud HTTP al servidor de recursos:

```
public class ProductController {  
    @Autowired  
    private WebClient webClient;  
  
    @GetMapping(value = "/products-view")  
    public List<Product> getProducts(  
        @RegisteredOAuth2AuthorizedClient("products-client-authorization-code") OAuth2AuthorizedClient  
        authorizedClient  
    ) {  
        return this.webClient  
            .get()  
            .uri("http://127.0.0.1:8090/products")  
            .attributes(oauth2AuthorizedClient(authorizedClient))  
            .retrieve()  
            .bodyToMono(new ParameterizedTypeReference<List<Product>>() {  
            })  
            .block();  
    }  
}
```


Consumir la vista del cliente

- Ahora, cuando ingresemos al navegador e intentemos acceder a la página **`http://127.0.0.1:8080/products-view`**, seremos redirigidos automáticamente a la página de inicio de sesión del servidor OAuth en **`http://auth-server:9000/login`** :



Please sign in

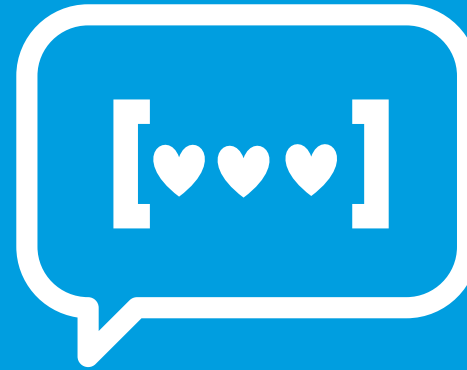
Username

Password

Sign in



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time
and effort you have taking in answering all of that is
important in order to improve our training plans so that you
will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us:

