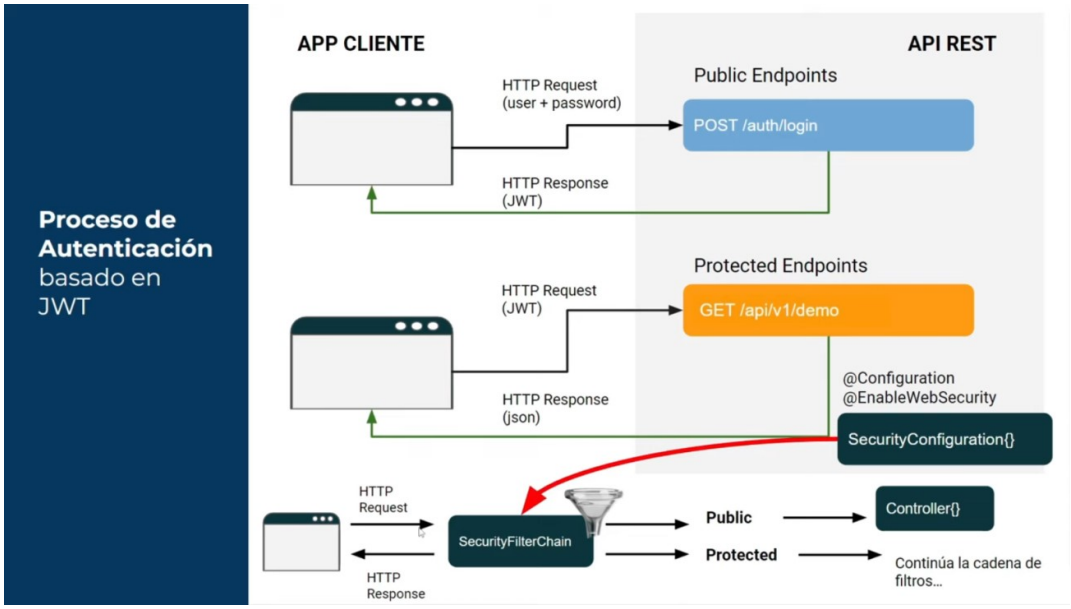
Como crear el Login con Spring Boot 3 + Spring Security 6 + JWT Authentication

Vamos a ver cómo crear el login desde la perspectiva desde la autenticación basada en JWT es decir Json Web Token. Vamos a estar trabajando con SPRING BOOT 3 y SPRING SECURITY 6.



La autenticación basada en JWT es un método de autenticación ampliamente utilizada en las API REST. Veamos cómo funcionan. La aplicación cliente es quien va a proporcionar las credenciales de inicio de sesión como ser el nombre de usuario y contraseña al servidor de autenticación en este caso lo va a realizar a través de un endpoint público como podemos ver en la imagen. La API REST va a verificar las credenciales y si estas son validas emitirá un JWT firmado. El mismo contiene información sobre la entidad del usuario y puede incluir otros datos relevantes como por ejemplo roles y permisos. El cliente va a recibir el JWT y lo va a almacenar localmente.

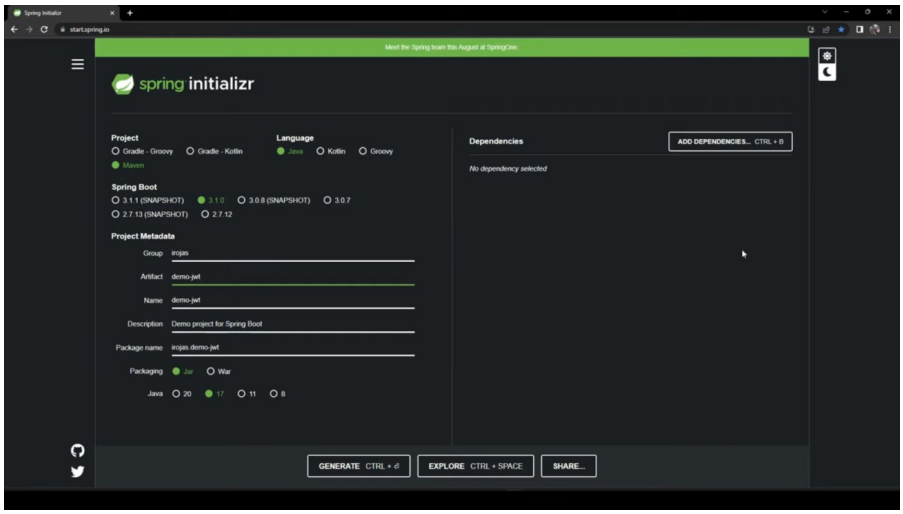
Generalmente en el almacenamiento de sesión o en alguna cookie segura. A continuación el cliente va a incluir este JWT en cada solicitud posterior que realice a fin de acceder a los recursos o a las funcionalidades que requiera. Por ello va a acceder a los endpoints protegidos, generalmente en el encabezado de la autorización de la petición HTTP. Finalmente el servidor de recursos en este caso la API REST va a verificar la autenticidad de ese JWT y si es válido va a procesar la solicitud y enviara la respuesta.

Antes de avanzar sobre el concepto de JWT es importante analizar cómo trabajar las rutas públicas y protegidas utilizando SPRING SECURITY. Para ello en primer lugar vamos a tener que añadir la dependencia de SPRING SECUTIRY a nuestro proyecto de SPRING BOOT.

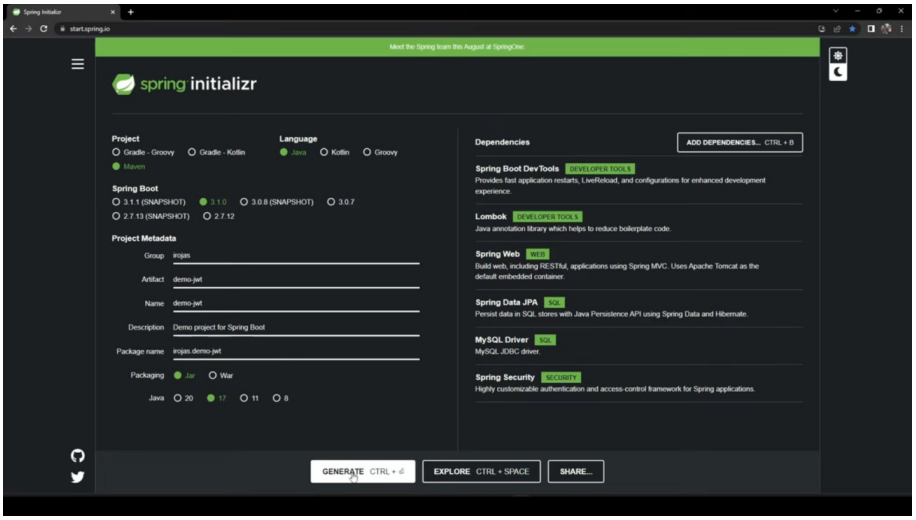
Luego debemos configurar el filtro securityFilterChain, sabemos nosotros que siempre va a pasar por un filtro. Esto lo podemos realizar con una clase de configuración que podemos llamarla SecurityConfiguration y lo importante aquí es que configuremos las anotation @Configuration @EnableWebSecurity.

Finalmente si la ruta es pública permitirá el acceso al recurso y caso contrario continuara con la cadena de filtros. En este caso nosotros vamos a ver como continuar la cadena de filtros teniendo en cuenta la autenticación basada en JWT.

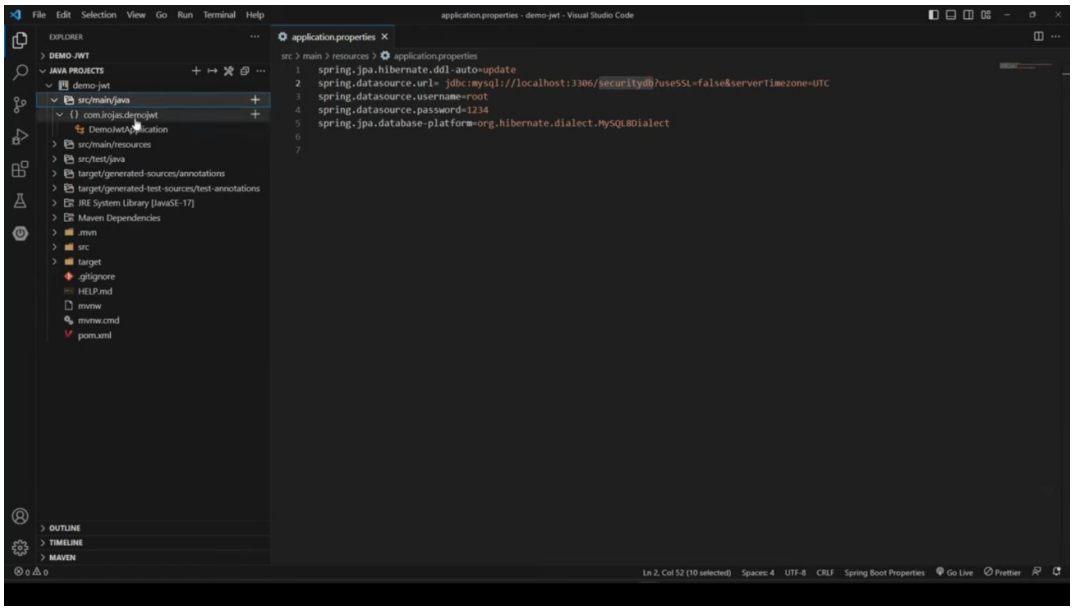
Veamos como configurar estos inicialmente antes de entender que significa el JWT.



Vamos al codeo. Vamos a comenzar creando nuestro proyecto de SPRING BOOT. En este caso vamos a estar trabajando con MAVEN, el Artifact lo vamos a llamar “demo-jwt” y vamos a dejar todo el resto por defecto y vamos a agregar las dependencias. Vamos a agregar la dependencia de DevTools, la dependencia de Lombok, la dependencia de SpringWeb, la dependencia de JPA porque vamos a estar trabajando con un usuario que va a estar en una base de datos por eso también necesitamos el driver de MySQL y por supuesto la dependencia de Spring Security.



Una vez que tengamos todo esto generamos nuestro proyecto, descomprimimos en algún lugar de nuestra computadora, en algún repositorio local para comenzar a trabajar. Y una vez que generamos nuestro proyecto lo abrimos en este caso con Visual Studio Code y vamos a application.properties y vamos a realizar las configuraciones relacionadas a la conexión con la base de datos, previamente también tenemos que crear la base de datos. En este caso ya la tenemos creada y la llamamos "securitydb" así que con eso creamos la conexión con la base de datos.



Tenemos que copiar este código en application.properties

```
spring.datasource.url:jdbc:mysql://localhost:3306/securitydb?allowPublicKeyRetrieval=true&useSSL=false&
useTimezone=true&serverTimezone=GMT&characterEncoding=UTF-8
spring.datasource.username: root
spring.datasource.password: root
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.jpa.show-sql:true
spring.jpa.hibernate.ddl-auto: update
spring.thymeleaf.cache: false
```

Ahora lo que vamos a hacer es **crear nuestro primer paquete**. Nuestro primer paquete va a contener el controlador para la autenticación por lo cual estos endpoints que van a permitir hacer el login y el registro de usuario deberían ser públicos y eso es lo que vamos a buscar. Vamos a crear un paquete que lo vamos a llamar "Auth", dentro de este paquete vamos a **crear una clase** que vamos a llamar "AuthController".

Recordemos que los controladores son los que nos van a permitir exponer las rutas con los endpoints. Vamos a agregar las anotation @RestContoller y @RequestMapping en este caso la ruta la vamos a especificar como "/auth" y obviamente utilizando Lombok vamos a utilizar esta anotation @RequiredArgsConstructor para hacer obligatorio que se agregue el constructor con todos los argumentos.

Vamos a continuación a especificar **los métodos** inicialmente lo vamos a hacer sencillos simplemente para poder hacer las pruebas. Por supuesto vamos a agregar las anotation @PostMapping para especificar las rutas. Lo mismo va a pasar con el registro.

Quedaría de la siguiente manera:

```java
package com.elavincho.demojwt.Auth;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.RequiredArgsConstructor;

@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    @PostMapping(value = "login")
    public String login(){
        return "Login from public endpoint";
    }
    @PostMapping(value = "register")
    public String register(){
        return "Register from public endpoint";
    }
}
```

Entonces ya tenemos nuestro controlador de autenticación en el cual va a exponer estos dos métodos.

A continuación otra cosa que **tenemos que crear es otro paquete** donde vamos a tener los endpoint que van a estar protegidos. Entonces vamos a crear en este caso un paquete que lo vamos a llamar **"Demo"** y **vamos agregar una clase** y lo vamos a llamar **"DemoController"** acá irían específicamente la API que ustedes hubieran creado y simplemente vamos a trabajar el controller por ahora y vamos a agregar las anotation pertinentes.

Quedaría de la siguiente manera:

```java
package com.elavincho.demojwt.Demo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.RequiredArgsConstructor;

@RestController
@RequestMapping("/api/v1")
@RequiredArgsConstructor
public class DemoController {

}
```

Una vez que tenemos las anotation **vamos a crear nuestro método** protegido. Agregamos la anotation @PostMapping y le vamos a especificar la ruta.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Demo;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.RequiredArgsConstructor;

@RestController
@RequestMapping("/api/v1")
@RequiredArgsConstructor
public class DemoController {

    @PostMapping(value = "demo")
    public String welcome(){
        return "Welcome from secure endpoint";
    }
}
```

Ya tenemos entonces con los endpoint que necesitamos para realizar las pruebas. Recordemos que como nosotros tenemos la dependencia de Spring Security por defecto lo que va a hacer es agregar la seguridad a todo los endpoints, es decir a todas las rutas que en este caso todas estarían protegidas. Por lo cual necesitamos realizar las configuraciones relacionadas a los filtros, es decir vamos a tener que agregar un nuevo paquete para configurar estos filtros iníciales.

Vamos a crear un nuevo paquete, en este caso lo vamos a llamar **"Config"** y dentro de este paquete **vamos a crear una clase** que la vamos a llamar **"SecurityConfig"**. Esta clase es la que **va a contener esa cadena de filtros** y ese **método securityFilterChain** que hablábamos anteriormente. Pero antes de eso no tenemos que olvidarnos de algunas anotation súper importantes que tienen que estar. Por un lado tiene que estar la anotation @Configuration. Esta anotation indica que esta clase es de configuración es decir que **va a tener métodos** que van a estar anotados **con la anotation @Bean** o marcados con la anotation @Bean los cuales se van a utilizar para configurar y crear los objetos que vamos a requerir en nuestra aplicación. También necesitamos la anotation @EnableWebSecurity y por supuesto hay que agregar la del constructor @RequiredArgsConstructor.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
import lombok.RequiredArgsConstructor;
```

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

}
```

Una vez hecho todo esto lo siguiente es trabajar nuestro método securityFilterChain que es lo que va a hacer es tener toda esa cadena de filtros que se va a ir ejecutando. En primer lugar vamos a trabajar la configuración relacionada a los endpoinds que van a estar públicos y diferenciar de los que van a estar protegidos.

Entonces vamos a **crear nuestro método** a fin de restringir el acceso a las rutas. Para eso vamos a crear un método que va a devolver un objeto **securityFilterChain**, lo vamos a llamar con el mismo nombre y lo que va a recibir por parámetro es un httpSecurity. No nos olvidemos de especificarlo como @Bean, así se puede luego crear el objeto y lo que vamos a hacer a continuación es retornar el http siempre y cuando pase por una serie de filtros. Acá vamos a configurar esa serie de filtros. El primer filtro que nosotros vamos a estar trabajando tiene que ver con las rutas privadas y protegidas como habíamos visto. En este caso **vamos a utilizar la expresión de Lamda** para agregar más de una configuración a la vez. Lo primero que vamos a hacer es especificar que el request machee con una ruta, nosotros dijimos que todos los request que machean con la ruta "Auth" como ser el login y el registro van a ser públicos. Es decir que yo debería tener acceso. Entonces le vamos a especificar "permit.All" y cualquier otro request le vamos a pedir que se autentique. Eso es lo que estamos haciendo en este punto.

A continuación vamos a llamar al formulario de login en este caso el que nos provee Spring Security con las configuraciones por defecto. Vamos a agregar las importaciones necesarias y a continuación vamos a llamar al método built(). Y no nos olvidemos de deshabilitar la protección "csrf" que habilita por defecto Spring Security. Y que significa esto. La protección "csrf" que significa "Cross-Site Request Forgery" es una medida de seguridad que se utiliza para agregar a las solicitudes Post una autenticación basada en un token "csrf" valido. Pero como nosotros no vamos a estar trabajando con esto lo vamos a deshabilitar. Si no nos los va a estar solicitar al momento de hacer las peticiones Post y nosotros vamos a trabajar en la autenticación basada en token pero en un token que vamos a estar creando nosotros. Es importante tener en cuenta que **este método lleva una throws Exception.**

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
```

```java
import org.springframework.security.web.SecurityFilterChain;
import static
org.springframework.security.config.Customizer.withDefaults;

import lombok.RequiredArgsConstructor;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception{
        return http
        .csrf(csrf ->
        csrf
        .disable())
        .authorizeHttpRequests(authRequest ->
            authRequest
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
            )
        .formLogin(withDefaults())
        .build();
    }
}
```

Ahora lo que queda es realizar las pruebas. Vamos a levantar nuestra aplicación de Spring Boot y a continuación vamos a realizar las pruebas en nuestros endpoints. Para eso vamos a estar trabajando con **Postman**. Vamos a crear una nueva petición. En este caso recordemos que los métodos eran Post y vamos a agregar nuestra ruta.

Recordemos que todas nuestras rutas que macheaban con Auth eran públicas que si hacemos clic en enviar, efectivamente esto me está dejando pasar y no me está pidiendo que inicie sesión, es decir las credenciales de usuario y password. Pero en el caso del demo que habíamos especificado que la ruta era /api/v1/demo en este caso no debería dejarme acceder y no debería mostrarme el mensaje. Vamos a probar y efectivamente esto está funcionando como corresponde porque me está mostrando el formulario de login. Habíamos dejado por defecto que nos mostrara el formulario de login en el caso de que el usuario debiera autenticarse porque todavía no hemos agregado las configuraciones relacionadas de autenticación basadas en JWT de hecho si agrego una nueva petición acá en el navegador podemos ver efectivamente ese formulario. Si escribimos las credenciales recordemos que esto es la configuración por defecto de Spring Security el usuario es “user” y la contraseña la tenemos en la terminal. Y si la colocamos nos va a dejar ingresar pero como no tenemos ningún HTML cargado nos va a dar error. En este caso nos da un error por que **cuando hacemos peticiones desde el navegador estamos utilizando el método “Get”** y recordemos que el método que necesitaba era el **Post**. Pero evidentemente con esto ya

ha quedado comprobado que las peticiones efectivamente gracias a los filtros que hemos configurado en securityFilterChain nos va a permitir acceder a algunos endpoints de manera pública y el resto van a quedar protegidos por lo cual en este caso requiere un formulario de login después modificaremos estos para que solicite el token y todo lo que tiene que ver con la autenticación basada en JWT.

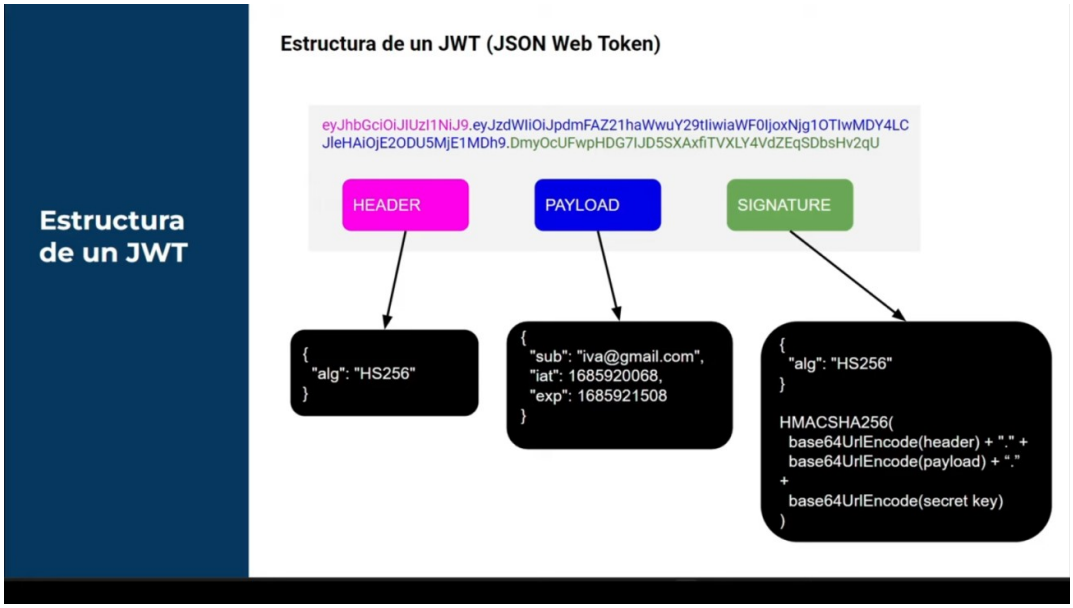
Estructura de un JWT (Json Web Token)

El JWT no es más que una cadena de caracteres que consta de tres partes separadas por un punto. El encabezado, la carga útil y la firma.

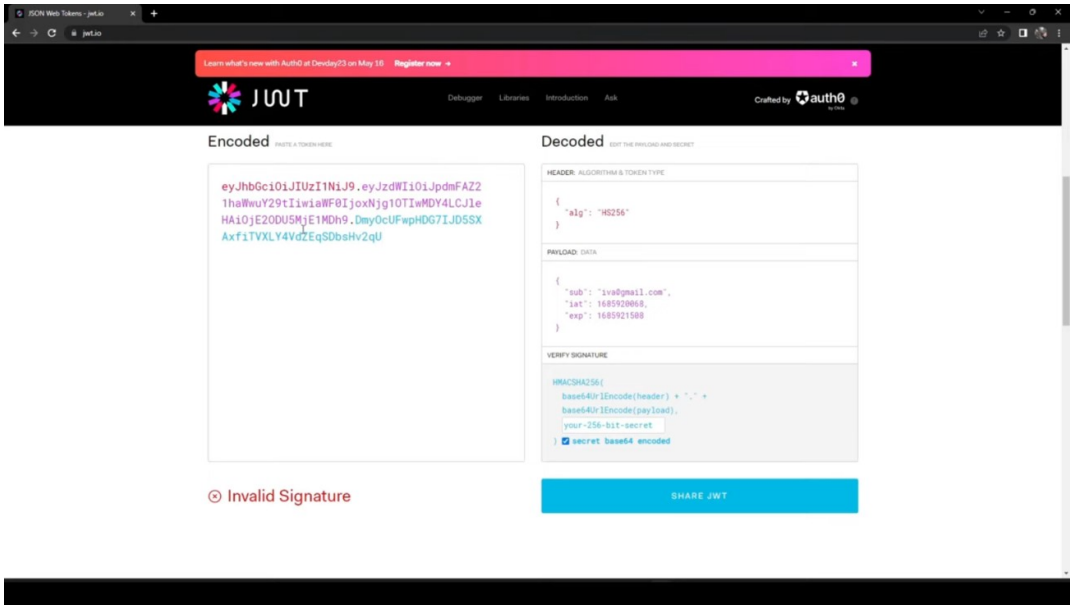
El encabezado contiene información sobre el tipo de token y el algoritmo de firma utilizado.

La carga útil contiene los datos de sesión como el identificador de usuario y los roles y permisos. También puede contener cualquier otra información que se quiera agregar.

La firma se utiliza para verificar la integridad del token y garantizar que no haya sido manipulado.

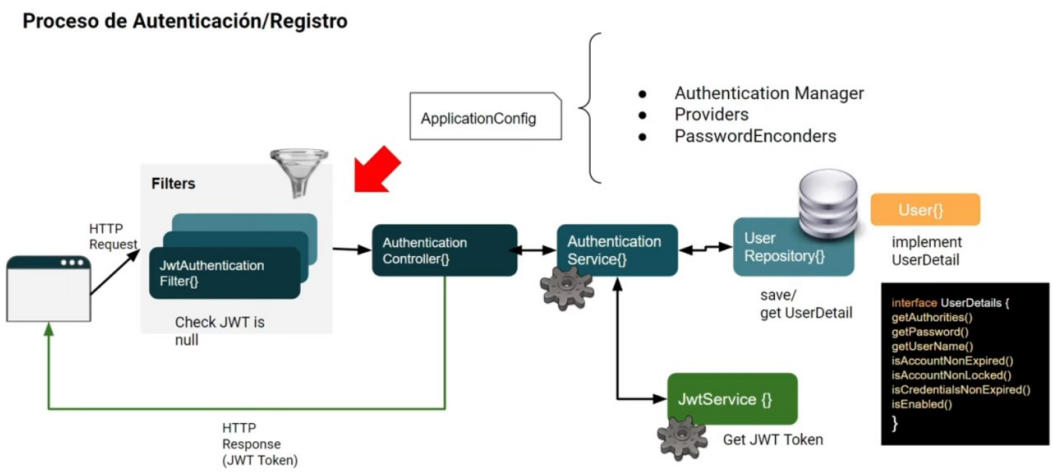


Es importante en este punto tener en cuenta que JWT es un token de autenticación **NO** de autorización, es decir que el JWT verifica la entidad del usuario pero no garantiza que tengas acceso a los recursos solicitados. La autorización debe ser implementada en el servidor de recursos. Si quieres evaluar tu JWT para verificar que este bien se puede utilizar la pagina jwt.io, en esta página se puede observar como el JWT es decodificado. Podemos ver en este caso el encabezado, el payload que sería la carga útil y la firma.

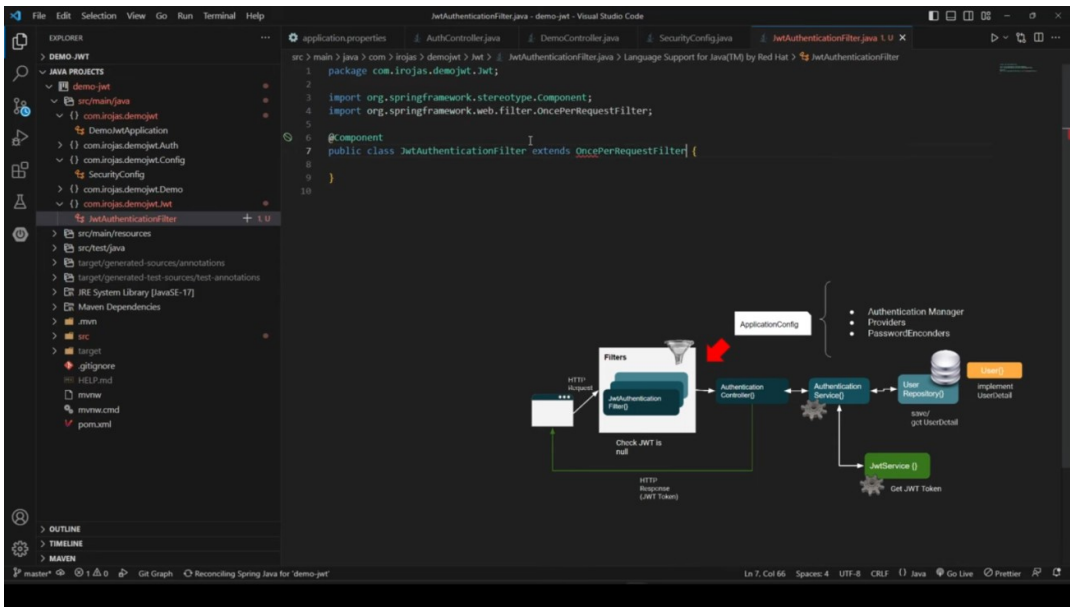


Proceso de Autenticación y Registro

Como todo proceso de autenticación va a comenzar con una petición http y como sabemos debe llamar al controller y se ejecutara un filtro. En este caso vamos a trabajar con el filtro JwtAuthenticationFilter, el mismo se va a ocupar de realizar todas las validaciones respecto al token. En este caso va a verificar que el mismo sea nulo, si el mismo es nulo el filtro concluye sin problemas y da lugar al AuthenticationController. El controlador entonces va a invocar al servicio de autenticación quien para el caso del registro de usuario va a guardar el nuevo registro en la base de datos. **Nota Importante** que la entidad "User" **debe obligatoriamente implementar la interface userDetails.** Por su parte para el login el servicio de autenticación buscará en la base de datos el usuario correspondiente. Finalmente generará el token y lo devolverá al AutenticationController que a su vez devolverá la respuesta al cliente. Ahora con el token en el cuerpo del mensaje. Importante destacar en este momento dado que no vamos a estar utilizando las configuraciones por defecto de Spring Security que deberemos configurar en nuestra aplicación de Spring Boot el proveedor correspondiente y el algoritmo de password encoder a utilizar. Debemos recordar que Spring Boot nos provee muchos escenarios para trabajar la autenticación.



Veamos a continuación como trabajar todo esto en nuestra aplicación de Spring Boot. Para esto **vamos a comenzar creando un nuevo paquete** que lo vamos a llamar **"Jwt"** el mismo va a tener obviamente todo lo relacionado a JWT y dentro de este paquete vamos a **crear una clase** que es la que tiene que ver con el filtro, entonces lo vamos a llamar **"JwtAutenticationFilter".**



No debemos olvidarnos de agregar la anotation @Component y extender de la clase abstracta OncePerRequestFilter. Esta clase abstracta se utiliza para crear filtros personalizados la razón del porque vamos a extender de esta clase es para garantizar que el filtro se ejecute solo una vez por cada solicitud http. Incluso si hay múltiples filtros dentro de la cadena de filtros. Como podemos observar nos muestra un error justamente porque tenemos que implementar los métodos. En este caso **el método que tenemos que implementar es el doFilterInternal.**

Este método es el que va a realizar todos los filtros relacionados al token. Entonces lo primero que tenemos que hacer en este caso es obtener el token de el request, **podemos observar que tenemos acceso al request, al response y al filterChain.** El filterChain recordemos que es la cadena de filtros que habíamos configurado anteriormente. Después vamos a volver sobre esa cadena de filtros un poco más hacia el final, dado que vamos a tener que agregar a esa cadena de filtros, el filtro relacionado al token. Para que encaje y continúe la cadena.

Entonces en este caso lo primero que tenemos que hacer es obtener el token entonces vamos a crear una variable que la vamos a llamar "token" y la vamos a obtener de **un método** que lo vamos a llamar **"getTokenFromRequest"** y obviamente **va a recibir como parámetro el request.** Ahora vamos a implementar ese método.

Pero habíamos dicho si el token es nulo le vamos a devolver a la cadena de filtros el control por así decirlo y vamos a retornar. Después vamos a ver qué sucede cuando tenemos el token, pero en este caso no lo vamos a tener porque vamos a estar trabajando con el registro y la autenticación, finalmente vamos a llamar nuevamente al filtro para que siga su curso.

A continuación entonces vamos a implementar nuestro método getTokenFromRequest, este método recordemos que nos va a devolver el token y por supuesto que devuelve un String porque el token no es más que una cadena de caracteres. Requiere también que le pasemos por parámetro el request justamente porque en el encabezado del request es donde nosotros vamos a obtener el token.

Entonces vamos a **crear una variable de tipo String** que la vamos a llamar **AuthHeader**, es decir primero vamos a tratar de encontrar del encabezado el ítem o la propiedad de autenticación. Entonces sabemos que está en el request, **vamos a llamar al método getHeader** y lo que nos interesa es el ítem de autenticación, entonces **vamos a trabajar con el HttpHeaders** y acá **nos interesa la autenticación**.

Este encabezado de este String que nosotros vamos a acceder va a comenzar con la palabra **"Bearer"** que estamos trabajando con la autenticación basada en token, entonces **lo primero que tenemos que hacer es verificar eso para retornar el token.**

Porque vamos a tener que extraer el token de esa cadena de caracteres sin incorporar la palabra "Bearer". Entonces vamos a tener que acceder a una estructura de control para verificar esto.

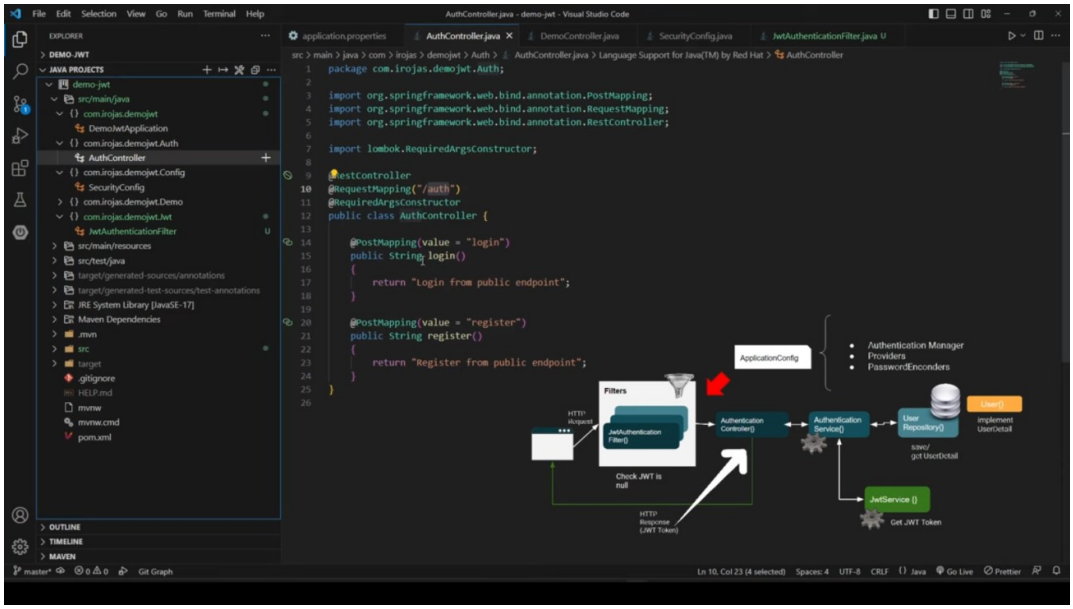
Entonces vamos a estar utilizando la librería de Spring Boot "StringUtils" para saber si existe el texto en el encabezado y además para evaluar que el AutHeader comience con la palabra "Bearer" y si esto es correcto entonces significa que a continuación de la palabra "Bearer" viene el token, entonces lo tenemos que extraer y retornar.

Entonces en este caso vamos a retornar autHeader.substring(7) en este caso vamos a especificar que a partir del carácter 7 hasta el final es el token. Caso contrario vamos a retornar null.

Quedaría de la siguiente manera:

```java
package com.elavincho.demojwt.Jwt;

import java.io.IOException;

import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class JwtAutenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        final String token = getTokenFromRequest(request);

        if (token==null){
            filterChain.doFilter(request, response);
            return;
        }

        filterChain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest request) {
        final String
authHeader=request.getHeader(HttpHeaders.AUTHORIZATION);

        if(StringUtils.hasText(authHeader) &&
authHeader.startsWith("Bearer ")){ //Notar que al final de la palabra
"Bearer " hay un espacio
            return authHeader.substring(7);
        }
        return null;
    }
}
```

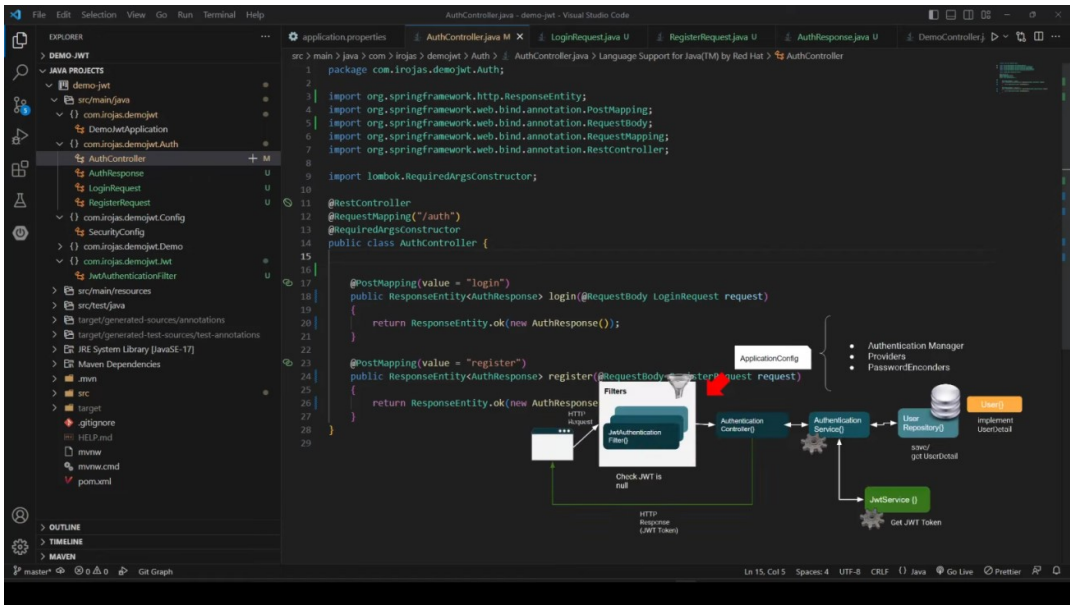
Una vez que tenemos configurado nuestro JwtAutenticationFilter **vamos a ir a nuestro controller de autenticación.**



Importante en este punto que vamos a estar trabajando las rutas que tienen que ver con el login y el registro y especifiquemos como va a esperar en este caso el controller los request y response. Para eso **vamos a crear tres clases**. **Vamos al paquete "Auth". Una la vamos a llamar LoginRequest** y la misma va efectivamente a pedir las credenciales, por lo tanto vamos a definir los atributos "username" y "password". **Importante** en este punto **agregar las anotation** relacionadas a Lombok, en este caso vamos a estar trabajando con **@Data** que va a permitir crear los getter y setter de manera automática, deja el código súper limpio. También **@Builder** para poder construir después los objetos de una manera también muy limpia. Por último lo relacionados a los constructores **@AllArgsConstructor** y **@NoArgsConstructor.** Al usar estas anotation dado que nos van a permitir mantener el código no solo más limpio sino que si agregamos por ejemplo un nuevo atributo no tengo que estar creando las propiedades getter y setter y tampoco tenemos que estar modificando los constructores que reciben todos los parámetros. Vamos a **crear** entonces también además **la clase relacionada con el registro**, la vamos a llamar **RegisterRequest**. Agregamos los atributos "username", "password", "firstname", "lastname", "country". No debemos olvidar **agregar las anotation @Data @Builder @AllArgsConstructor** y **@NoArgsConstructor.** Ya tenemos nuestra para el request en caso del registro y vamos a **crear otra clase para la respuesta.** En este caso **la vamos a llamar AuthResponse**. Justamente porque va a ser la respuesta independientemente si es el registro o es el login. **Nos interesa que nos devuelva el token**. Por ende el atributo que necesitamos es justamente el "token" y recordemos que no es más que una cadena de caracteres. A continuación vamos a **agregar las anotation** de Lombok **@Data @Builder @AllArgsConstructor** y **@NoArgsConstructor.** Perfecto ya tenemos entonces nuestra tres clases que van a requerirse para acceder a las peticiones.

Volvemos a la clase AuthController, lo que vamos a hacer a continuación es configurar nuestros endpoint de login y response. Importante en este punto hablar un poco sobre el **concepto de ResponseEntity.** Es el objeto que nosotros vamos a estar devolviendo. El ResponseEntity básicamente va a representar todas las respuestas http, va a incluir los códigos de estado, los encabezados y el cuerpo de respuesta. Es por ello que vamos a estar utilizando esta clase dado que nos proporciona flexibilidad para

configurar y personalizar la respuesta http. En este caso le vamos a decir que **la respuesta va a ser del tipo AuthResponse** como acabamos de definir previamente en la clase. En el cuerpo del mensaje vamos a estar accediendo a **las credenciales del usuario que están definidas en nuestra clase del LoginRequest**. Por supuesto que esto está dejando de compilar dado que ya no vamos a estar devolviendo una cadena de caracteres es decir un String y **vamos a estar devolviendo un objeto del tipo ResponseEntity.** Entonces vamos a utilizar la misma clase, **vamos a llamar al método ok()** si se supone que todo funcionó bien. En este caso **vamos a devolver una respuesta del tipo AuthResponse** como configuramos previamente. Por supuesto que acá va a estar devolviendo un objeto vacio por así decirlo, ya volveremos sobre esto. **Lo mismo vamos a hacer con el registro.** Recordemos nuestro esquema de procesamiento de autenticación y registro, **recordemos que el AutenticationController se va a comunicar con el AuthenticationService para acceder al token**, en el caso del login irá a buscar el usuario que esta autenticado y en el caso del registro se creará un nuevo registro de usuario.



Entonces **tenemos que agregar un nuevo atributo** que lo vamos a llamar **AuthService, por el cual vamos a acceder a los métodos del login y response** justamente **para acceder al token** que va a ser la respuesta y nos va a estar devolviendo. Por supuesto que va a requerir el request y lo mismo voy a realizar con el registro.

Pasamos de esto:

```
package com.elavincho.demojwt.Auth;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.RequiredArgsConstructor;

@RestController
```

```
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    @PostMapping(value = "login")
    public String login(){
        return "Login from public endpoint";
    }
    @PostMapping(value = "register")
    public String register(){
        return "Register from public endpoint";
    }
}
```

A esto:

```
package com.elavincho.demojwt.Auth;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import lombok.RequiredArgsConstructor;

@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthService authService;

    @PostMapping(value = "login")
    public ResponseEntity<AuthResponse> login(@RequestBody LoginRequest
request){
        return ResponseEntity.ok(authService.login(request));
    }
    @PostMapping(value = "register")
    public ResponseEntity<AuthResponse> register(@RequestBody
RegisterRequest request){
        return ResponseEntity.ok(authService.register(request));
    }
}
```

Esto no está funcionando porque efectivamente no tenemos el servicio con los métodos. Vamos a crearlos a partir de aquí. **Creamos la clase AuthService y el método login y register en la misma clase.** Esto aun no va a compilar porque los métodos los crea automáticamente como objetos y el AutenticationController está

esperando un objeto del tipo AuthResponse. Simplemente vamos a modificar los métodos en la clase AuthService.

Pasamos de esto:

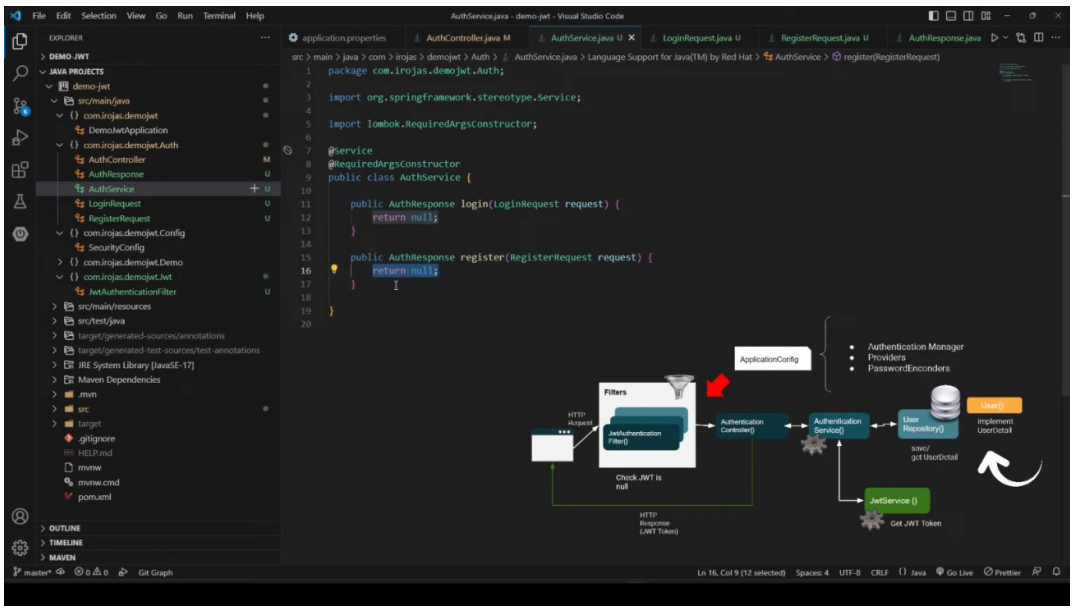
```
package com.elavincho.demojwt.Auth;

public class AuthService {

    public Object login(LoginRequest request) {
        return null;
    }

    public Object register(RegisterRequest request) {
        return null;
    }

}
```

A esto:

```
package com.elavincho.demojwt.Auth;

import org.springframework.stereotype.Service;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class AuthService {

    public AuthResponse login(LoginRequest request) {
        return null;
    }

    public AuthResponse register(RegisterRequest request) {
        return null;
    }

}
```

Y ahora ya debería estar compilando.

No nos olvidemos en el servicio de **agregar las anotation** relacionada al servicio **@Service.** También hay que agregar las anotation para el constructor **@RequiredConstructor.** Una vez hecho esto **vamos a empezar** en este caso por una cuestión de lógica **a configurar el método de registro. Importante** en este caso **tener el modelo de usuario y el repositorio** porque lo primero que tenemos que hacer es crear el usuario.



Una vez que creamos el usuario vamos a invocar al servicio de token o JwtService para acceder al nuevo token que se va a generar de manera automática. Entonces lo primero que vamos a hacer es **crear nuestro modelo**. En este caso vamos a trabajar con **un nuevo paquete** que lo vamos a llamar **User** y acá vamos a tener configurado todo lo relacionado al usuario. Vamos a **crear una** primera **clase** que va a ser el rol del usuario. La vamos a llamar **"Role"** pero **en este caso no va a ser una clase, va a ser un enum** para que sea más sencillo. Hay que crear dos perfiles de usuario, por un lado el administrador y por otro lado el usuario.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.User;

public enum Role {
    ADMIN,
    USER
}
```

A continuación vamos a **crear** nuestra **clase** de usuario, vamos a llamarla **"User"** y **vamos a agregarle los atributos** "id", "username", "lastname", "firstname", "country", "password" recordemos que van a ser los mismos que vamos a especificar en el registro, además le **vamos a agregar el rol "role"** como atributo. Una vez que tenemos esto lo que tenemos que hacer a continuación es **agregar las anotation** de Lombok para poder manipular en este caso los objetos fácilmente y como sabemos nosotros vamos a estar trabajando con JPA por lo cual también tenemos que agregar la anotation de **@Entity**. También tenemos que **agregar las anotation relacionados al mapeo.** Tenemos que especificar que "id" corresponde al identificador **@Id** y además queremos que se genere de manera automática, por eso vamos a agregar la anotation **@GeneratedValue**. Es **importante especificar que la tabla "user"** va a tener en este caso como **"uniqueConstraints"** es decir no se puede repetir el username. Finalmente vamos a **agregar la anotation @Column** y vamos a especificar que esta columna

(username) no sea nula. Es decir a nivel de base de datos cuando se crea el mapeo que es lo que estamos haciendo con esta anotation es que no va a permitir realizar ningún insert a la columna si el dato de username esta vacio o nulo. Con esto ya completamos todo lo que tiene que ver con el mapeo con que es específico de JPA, **nos está faltando el repositorio** que ya lo vamos a ver a continuación. Pero es **importante** no olvidarnos **realizar la implementación de UserDetails** porque sino no vamos a poder trabajar con la autenticación. Entonces **debemos en nuestra clase de User implementar UserDetails.** Por supuesto esto va a dejar de compilar, recordemos que **las interfaces son como un contrato que nos obliga a implementar ciertos métodos. Agregamos los métodos** que tenemos que estar implementando. Todos los **métodos que devuelven un booleano** le indicamos que **retornen True** y como nosotros vamos a estar trabajando con un token y el mismo token es el que de alguna manera va a especificar cuando expiró porque tiene un tiempo de caducidad, esto lo vamos a revisar después en el servicio de JWT. Dicho esto nos va a quedar solamente **el método GrantedAuthority** y lo que **vamos a** hacer acá es **retornar una lista que contiene un único objeto** que va a representar de alguna manera la autoridad otorgada al usuario autenticado. Por ello es importante también especificar el rol.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.User;

import java.util.Collection;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import jakarta.persistence.UniqueConstraint;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Entity
```

```java
@Table(name="user", uniqueConstraints = {@UniqueConstraint(columnNames =
{"username"})})
public class User implements UserDetails{
    @Id
    @GeneratedValue
    Integer id;
    @Column(nullable = false)
    String username;
    String lastname;
    String firstname;
    String country;
    String password;
    @Enumerated(EnumType.STRING)
    Role role;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority((role.name())));
    }
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

Vamos a crear a continuación y ya para completar **el repositorio**. **Creamos** en **el paquete User la clase UserRepository.** Recordemos que **NO** es una clase, **tiene que ser un interface** y **debe extender de JpaRepository. Importante especificar la clase** que vamos a tener **mapeada** es decir nuestro modelo **y el tipo de dato** que corresponde al identificador **<User, Integer>.** Una vez que tenemos esto ya tenemos implementado nuestro repositorio. También obviamente que en este caso si recuerdan **Jpa** nos va a proveer los métodos para hacer un **CRUD** básico a la base de datos. Pero en algunos casos vamos a requerir algunos otros métodos más específicos. Que son los llamados **Query métodos**. **Vamos a crear un Query método específico para buscar por username.** A continuación vamos a especificar que el método **va a devolver un objeto del tipo <User>** pero este va a ser **Optional**. Es decir que en su caso podría devolver un null. Luego vamos a especificar **el nombre del Query método**, en este caso va a ser **"FindByUsername"** recuerden que hay reglas de cómo escribir los Query métodos.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.User;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Integer> {

    Optional<User> findByUsername(String username);

}
```

Dicho esto **vamos a ir** a nuestro servicio de autenticación **a la clase AuthService.** Y vamos a empezar **configurando el registro de usuario.** Lo primero que tenemos que hacer es **crear un objeto del tipo User** y es **importante importar de la clase que nosotros hemos creado** y **NO** del usuario propio de Spring Boot. En este caso **vamos a usar el patrón de diseño** en este caso **builder()** para la construcción de objetos y vamos a llamar en este caso al atributo username. Vamos a ir cargando cada uno de los atributos. **El username recordemos que nosotros lo tenemos en request** de la petición http y también recordemos que **habíamos creado una clase que nos iba a permitir acceder a los getter y setter** gracias a que hemos agregados esas anotation de Lombok. Por lo cual podemos acceder directamente escribiendo **request.** (punto) y **podemos acceder al método getUsername().** Lo mismo vamos a hacer con el resto de los parámetros. En el caso del rol vamos a estar trabajando con la enumeración y cuando se cree el usuario por primera vez, va a ser del tipo usuario. Una vez que ya tenemos nuestro objeto de usuario vamos a tener que invocar al repositorio para pasarle el objeto en este caso de usuario para que se cree un nuevo registro en la base de datos.

Entonces vamos a **crear una variable para el repositorio** y una vez que tenemos eso vamos a llamar al repositorio y vamos a **invocar el método save()** y le vamos a pasar por **parámetro el objeto de usuario**. Una vez que tenemos esto **se va a insertar el nuevo registro en la base de datos.** A continuación como debemos **devolver un objeto del tipo AuthResponse** vamos a **retornar un objeto de este tipo** y vamos a trabajar con el **patrón de diseño builder()** para la construcción del mismo y evidentemente **tenemos que pasarle el token** que debemos generar. Luego vamos a **llamar al método build()** para terminar de construir el objeto y aquí tenemos la respuesta y por ende ya debería compilar. Y recordamos nuestro esquema con el flujo de trabajo o el workFlow.

Pasaríamos de esto:

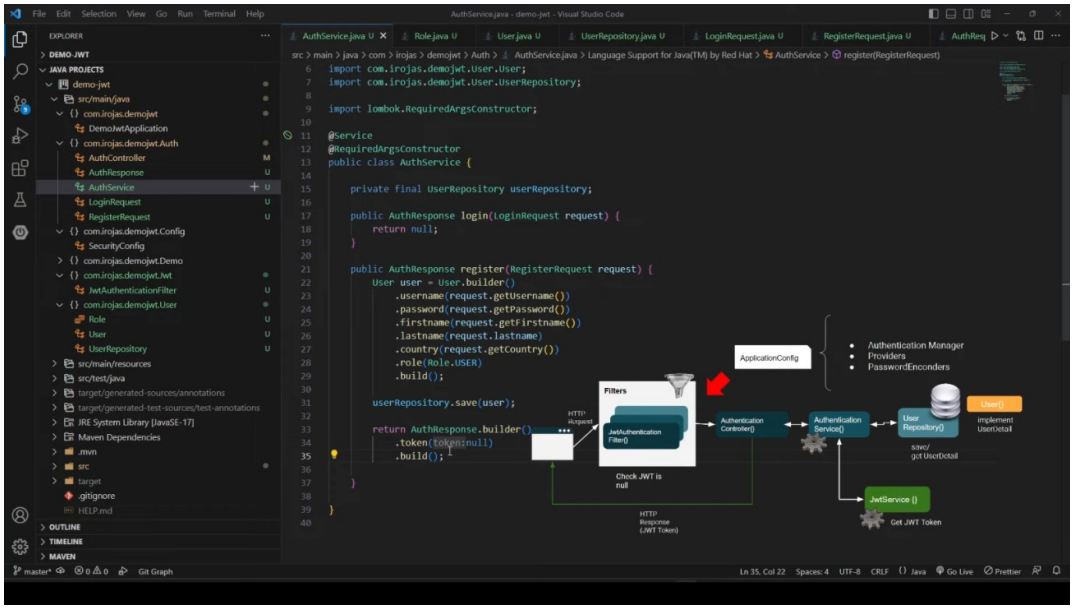
```
package com.elavincho.demojwt.Auth;

import org.springframework.stereotype.Service;

import lombok.RequiredArgsConstructor;
```

```java
@Service
@RequiredArgsConstructor
public class AuthService {

    public AuthResponse login(LoginRequest request) {
        return null;
    }
    public AuthResponse register(RegisterRequest request) {
        return null;
    }
}
```

A esto:

```java
package com.elavincho.demojwt.Auth;
import org.springframework.stereotype.Service;
import com.elavincho.demojwt.User.Role;
import com.elavincho.demojwt.User.User;
import com.elavincho.demojwt.User.UserRepository;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository userRepository;

    public AuthResponse login(LoginRequest request) {
        return null;
    }

    public AuthResponse register(RegisterRequest request) {
        User user = User.builder()
            .username(request.getUsername())
            .password(request.getPassword())
            .firstname(request.getFirstname())
            .lastname(request.getLastname())
            .country(request.getCountry())
            .role(Role.USER)
            .build();

        userRepository.save(user);

        return AuthResponse.builder()
            .token(null)
            .build();
    }
}
```

Y recordamos nuestro esquema con el flujo de trabajo o el workFlow, **el servicio va a tener que comunicarse con otro servicio** que lo vamos a llamar **JwtService** el cual **se va a encargar** de todo lo relacionado al **token**.



En este caso por ahora lo único que necesitamos que este token se genere de manera automática. Para eso **vamos a crear** nuevamente **el servicio** que lo vamos a llamar **JwtService** y luego **vamos a invocar un método** al cual vamos a estar llamando a este servicio. Le vamos a poner como nombre **getToken** y le **vamos a mandar** por **parámetro** nuestro **objeto de usuario**.

```java
package com.elavincho.demojwt.Auth;

import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import com.elavincho.demojwt.Jwt.JwtService;
import com.elavincho.demojwt.User.Role;
import com.elavincho.demojwt.User.User;
import com.elavincho.demojwt.User.UserRepository;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository userRepository;
    private final JwtService jwtService;
    private final PasswordEncoder passwordEncoder;
```

```
    public AuthResponse login(LoginRequest request) {
        return null;
    }

    public AuthResponse register(RegisterRequest request) {
        User user = User.builder()
            .username(request.getUsername())
            .password(passwordEncoder.encode(request.getPassword()))
            .firstname(request.getFirstname())
            .lastname(request.getLastname())
            .country(request.getCountry())
            .role(Role.USER)
            .build();

        userRepository.save(user);

        return AuthResponse.builder()
            .token(jwtService.getToken(user))
            .build();
    }
}
```

Por supuesto que esto no va a compilar, para ello **vamos a crear** nuestra **clase JwtService** y también vamos a **implementar el método getToken**.

Vamos a ir a la clase JwtService recordemos que al ser un servicio tenemos que **agregar la anotation @Service** y también en este caso **tenemos que cambiar el objeto** que **necesita recibir** por **parámetro**. Vamos a trabajar con el **UserDetails**.

Como podemos observar esto igual queda compilando justamente porque nuestro objeto usuario implementa la interfaz UserDetails. Por lo cual esto va a funcionar sin problemas. **Sin embargo observamos que en este caso JwtService no los agrego en el paquete de autenticación y debería estar en el paquete Jwt** para tener un orden y simplemente la movemos.

Pasamos de esto:

```
package com.elavincho.demojwt.Auth;

import com.elavincho.demojwt.User.User;

public class JwtService {

    public String getToken(User user) {
        return null;
    }

}
```

A esto:

```
package com.elavincho.demojwt.Jwt;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

@Service
public class JwtService {

    public String getToken(UserDetails user) {
        return null;
    }
}
```

Ya estamos listos para comenzar a trabajar. Lo primero que tenemos que hacer es **ir a nuestro archivo pom.xml** y **agregar las dependencias relacionadas a JWT**. En este caso vamos a estar trabajando con la **biblioteca io** y particularmente todo lo relacionado JWT. Vamos a estar trabajando con los artefactos de **jjwt-api, jjwt-impl** y **jjwt-jackson.** También vamos a estar trabajando con la **versión 0.11.5.**

```
<dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-api</artifactId>
            <version>0.11.5</version>
        </dependency>

        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-impl</artifactId>
            <version>0.11.5</version>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-jackson</artifactId>
            <version>0.11.5</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

Dicho esto **vamos a ir a nuestra clase JwtService** y **vamos a generar el token**. Para eso **vamos a crear a otro método** que se va a llamar **getToken** pero **va a recibir por parámetro un HashMap**. Recordemos que un HashMap no es más que una **clase de colecciones** que se utiliza para almacenar pares de **clave-valor**. Nosotros lo vamos a estar utilizando en los **Claims** en nuestra aplicación para pasar información adicional en el token. Vamos a continuación a **crear nuestro método getToken** y lo único que

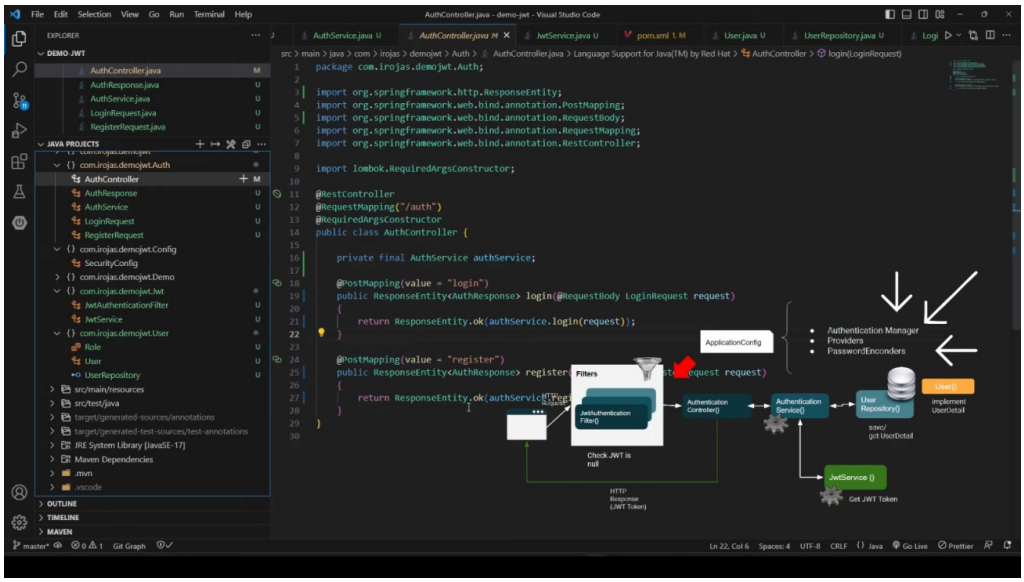
tenemos que hacer es **cambiar el Object** por un **Map**, donde tenemos que especificar los tipos de datos de los pares de clave valor. Va a ser un **String y un Object** y lo vamos a llamar **extraClaims. Agregamos** la referencia a **Map** (importamos) y ya tenemos todo listo para generar nuestro token. Para generar nuestro token **vamos a trabajar con la librería Jwts** en este caso lo que **vamos** a hacer es **construir el objeto** y le **vamos a estar seteando todos los datos** que requiera. En este caso por ejemplo los **Claims (setClaims)** que lo recibimos por parámetro. También vamos a estar seteando los **Subjects (setSubject)** que no es más que el username. Por supuesto y no puede faltar la **fecha de expiración** (setIssuedAt) en este caso **vamos a pasar primero la fecha en que se creo**, vamos a estar trabajando específicamente la de sistema y en este caso también la fecha de expiración (setExpiration). A esto le vamos a sumar un día. Vamos a **agregar** también **la importación Date** y finalmente **tenemos que pasar la firma (signWith)**, observamos acá distintas firmas para este método vamos a movernos **en el que está en el orden 2** y fíjense lo que necesita en un primer momento es un **objeto del tipo key y también el algoritmo**. El objeto del tipo key todavía no lo tenemos así que **vamos a crear un método** que lo vamos a llamar **getKey** y **vamos a especificar que algoritmo de encriptación** vamos a estar utilizando **(SignatureAlgorithm)** para trabajar la clave secreta. En este caso vamos a estar trabajando con **HS-256**. Y finalmente **vamos a llamar al método compact()** para que cree el objeto y lo serialice. En este caso al token. Finalmente tenemos que **crear** nuestro **método getKey** como podemos observar **va a devolver una key** y nosotros podemos especificar una variable estática del tipo String donde **vamos a especificar nuestra SECRET KEY**. Por supuesto que podes escribir la key que nosotros queramos. Ahora lo que tenemos que hacer es esta key que está en String llevarla a base 64 para mandarla como key a la firma de nuestro token. Para eso **vamos a trabajar con una array de bytes, vamos a llamar keyBytes** y **vamos a utilizar la clase Decoders.BASE64** para **decodificar** nuestra **SECRET KEY**. Finalmente vamos a retornar, para ello **vamos a llamar al método hmacShakeyFor** de la **clase key** que **nos va a permitir crear una nueva instancia** de nuestra SECRET KEY.

Quedaría de esta manera:

```
package com.elavincho.demojwt.Jwt;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;

@Service
public class JwtService {
```

```
    private static final String SECRET_KEY="123456789";

    public String getToken(UserDetails user) {
        return getToken(new HashMap<>(), user);
    }

    private String getToken(Map<String, Object> extraClaims, UserDetails
user) {
        return Jwts
        .builder()
        .setClaims(extraClaims)
        .setSubject(user.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new
Date(System.currentTimeMillis()+1000*60*60*24))
        .signWith(getKey(), SignatureAlgorithm.HS256)
        .compact();
    }

    private Key getKey() {
        byte[] keyBytes=Decoders.BASE64.decode(SECRET_KEY);

        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

En este punto podrías pensar que ya está todo listo, al menos el registro, dado que **si vamos a la clase AuthService** podemos observar que **se crea el objeto usuario**, **se guarda en la base de datos utilizando el repositorio** y finalmente **se obtiene el token** a través del **servicio de Jwt** y retorna en este caso al controlador quien lo va a retornar al cliente.



Pero esto aun no va a funcionar porque nosotros **tenemos que decirle específicamente a Spring Boot** cuál es el **Authentication Manager** que tiene que utilizar en cuanto a proveedor de en este caso tiene que ser un proveedor de acceso a datos y en el caso del password encoder tenemos que especificar cuál es el algoritmo que tiene que utilizar ese password encoder para finalmente codificar el password y llevar la encriptación en la base de datos. Para eso **vamos a ir a nuestro paquete de configuración** y vamos a **crear una nueva clase** que la vamos a llamar **AplicationConfig.** En primer lugar vamos a **agregar las anotation** de configuración **@Configuration** y la relacionada con el constructor **@RequiredArgsConstructor**, luego vamos a **crear un método que permita al manejador acceder a las instancias de el AuthenticationManager**, para eso vamos a **crear un método publico** que **va a permitir** acceder **a la instancia** del **AuthenticationManager** por lo cual **va a devolver un objeto del tipo authenticationMagener** y vamos a **recibir por parámetro** el **authenticationConfiguration** que lo vamos a llamar **config** el cual nos va a permitir acceder a la instancia y así **retornar a través del método getAuthenticationManager**.

Por supuesto que no va a dar error porque **tenemos que agregar la declaración de errores** y también tenemos que **agregar la anotation @Bean**. A continuación también necesitamos **crear un método** que devuelva el proveedor de, es decir el **AuthenticationProvider**, en este caso no va a recibir nada por parámetro y el AuthenticationProvider que nosotros vamos a estar trabajando es el **DaoAuthenticationProvider**. Vamos a crear una nueva instancia y finalmente **vamos a setear el userDetailsService** que por ahora no lo tenemos, así que **vamos a crear un método** que lo vamos a llamar **UserDetailsServices** y también **necesitamos setear** el **PasswordEncoder**. Tampoco lo tenemos. Vamos a **crear un método** que después lo vamos a llamar y finalmente **retornamos** la **instancia** de **authenticationProvider**. Obviamente a esto también tenemos que **agregarle la anotation @Bean**. A continuación vamos a **crear los métodos** que nos devuelvan tanto el **UserDetailsService** como el **PasswordEncoder**, esto también tenemos que anotarlos con la **anotation @Bean** para que el manejador pueda acceder a las instancias y en este caso **vamos a retornar** una nueva instancia y **vamos a trabajar** con **BCryptPasswordEncoder** y **no va a recibir ningún parámetro**.

En el caso del **UserDetailsServices** vamos a estar trabajando con una expresión de **Lamda** para poder acceder al username por supuesto que primero tenemos que buscar el username y **si** por alguna de esas cuestiones **no existe** entonces en ese caso **vamos a lanzar una excepción** del tipo **UserNameNotFoundException** y le vamos a **pasar un texto** que especifique esto. Esto aun **no está compilando porque el repositorio no lo tenemos**, pero la vamos a crear manualmente. Para eso **vamos a crear una variable** del tipo **UserRepository** que es la que nosotros vamos a estar requiriendo.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.AuthenticationProvider;
```

```java
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider
;
import
org.springframework.security.config.annotation.authentication.configurati
on.AuthenticationConfiguration;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

import com.elavincho.demojwt.User.UserRepository;

import lombok.RequiredArgsConstructor;

@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {
    private final UserRepository userRepository;
    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration config) throws
Exception{
        return config.getAuthenticationManager();
    }
    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authenticationProvider = new
DaoAuthenticationProvider();

authenticationProvider.setUserDetailsService(userDetailService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());

        return authenticationProvider();
    }
    @Bean
    private PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    private UserDetailsService userDetailService() {
        return username -> userRepository.findByUsername(username)
        .orElseThrow(()-> new UsernameNotFoundException("Usuario no
encontrado!"));
    }

}
```

Finalmente **no podemos olvidarnos** de nuestro **SecurityConfig** donde tenemos **configurado toda la secuencia de filtros**. **Vamos a la clase SecurityConfig.**

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
import org.springframework.security.web.SecurityFilterChain;
import static
org.springframework.security.config.Customizer.withDefaults;

import lombok.RequiredArgsConstructor;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception{
        return http
        .csrf(csrf ->
        csrf
        .disable())
        .authorizeHttpRequests(authRequest ->
            authRequest
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
            )
        .formLogin(withDefaults())
        .build();
    }
}
```

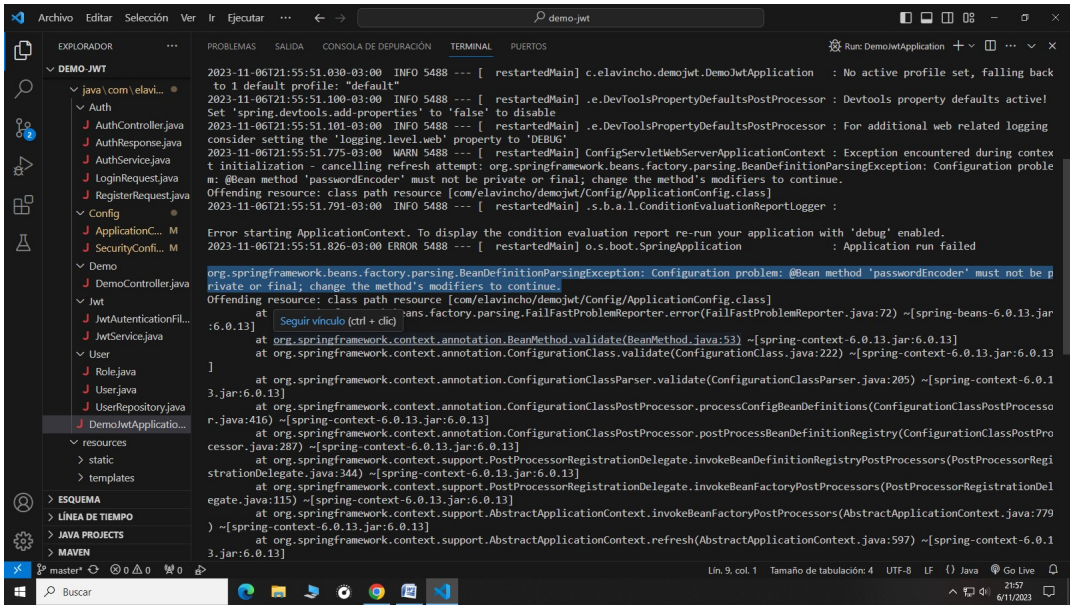
Como podemos observar y recordar **lo único que tenemos configurado son las rutas públicas y privadas** y en este caso estamos trabajando con la autenticación propia de Spring Security. Nosotros ahora **vamos a trabajar** con una **autenticación basada en Jwt** por lo cual vamos a requerir cambiar algunas cuestiones. En primer lugar **vamos a inhabilitar las sesiones** (sessionManagement), para ello vamos a utilizar la expresión de **Lamda** nuevamente dado que se deja leer más fácilmente. En este caso especificar la política de creación de sesión en este caso la vamos a poner que efectivamente no la utilice. También **vamos a especificar el AuthenticationProvider** que **lo vamos a crear** y **vamos a agregar el filtro relacionado a**

jwtAuthenticationFilter que configuramos previamente y **tenemos que especificar la clase**. Esto no está compilando, **tenemos que crear las variables que hacen falta (jwtAuthenticationFilter y authProvider)**. Ya tenemos nuestro filtro relacionado a la autenticación basada en token y finalmente el proveedor. Vamos a borrar esta dependencia que ya no se está utilizando (importación de withDefaults).

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.AuthenticationProvider;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
//import static
org.springframework.security.config.Customizer.withDefaults;
import
org.springframework.security.web.authentication.UsernamePasswordAuthentic
ationFilter;

import com.elavincho.demojwt.Jwt.JwtAutenticationFilter;

import lombok.RequiredArgsConstructor;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    public final JwtAutenticationFilter jwtAutenticationFilter;
    public final AuthenticationProvider authProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception{
        return http
        .csrf(csrf ->
        csrf
        .disable())
        .authorizeHttpRequests(authRequest ->
            authRequest
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
            )
```

```
        .sessionManagement(sessionManager ->
        sessionManager
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authenticationProvider(authProvider)
        .addFilterBefore(jwtAutenticationFilter,
UsernamePasswordAuthenticationFilter.class)
        .build();
    }
}
```

Y vamos a ver si esto está funcionando. Para eso vamos a levantar nuestra aplicación de Spring Boot, vamos a ver inicialmente si esto compila.



Evidentemente **no levanto** y esto **se debe a que el Password Encoder** en este caso **esta como privado o final**, tenemos que cambiar el modificar el modificador del método. **Vamos a la clase ApplicationConfig** y como el método lo habíamos creado por defecto se creó como privado. **El manejador necesita acceder por lo cual estos métodos tienen que ser públicos.** Entonces simplemente **cambiamos los métodos Password Encoder y UserDetailsServices a públicos y listo**. También **hay un error** en la **función AuthenticationProvider** que **está devolviendo el mismo método** y solo **debería retornar la instancia**, así que **debemos borrar los paréntesis** que tiene el **AuthenticationProvider**() del retorno.

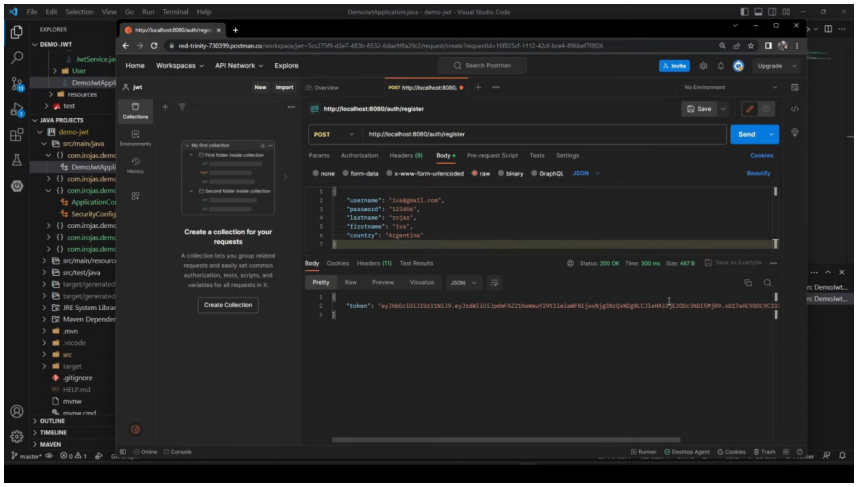
Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
```

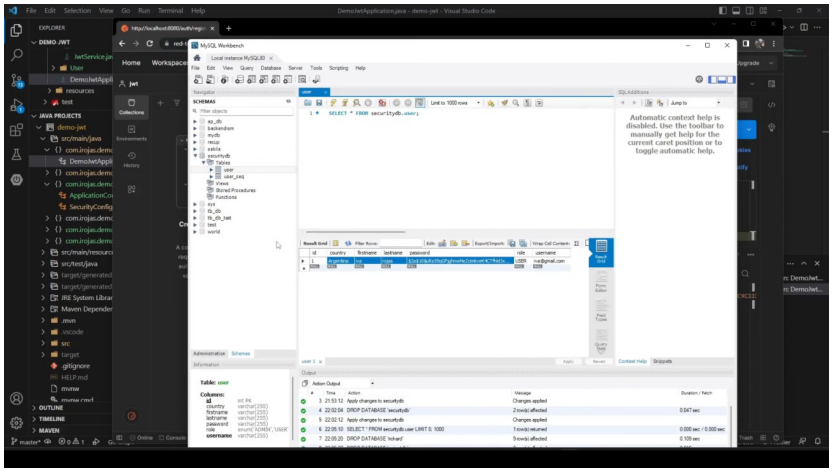
```java
import
org.springframework.security.authentication.AuthenticationProvider;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider
;
import
org.springframework.security.config.annotation.authentication.configurati
on.AuthenticationConfiguration;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

import com.elavincho.demojwt.User.UserRepository;

import lombok.RequiredArgsConstructor;

@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {

    private final UserRepository userRepository;

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration config) throws
Exception{

        return config.getAuthenticationManager();

    }

    @Bean
    public AuthenticationProvider authenticationProvider(){

        DaoAuthenticationProvider authenticationProvider = new
DaoAuthenticationProvider();

authenticationProvider.setUserDetailsService(userDetailService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());

        return authenticationProvider;
    }

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

```
    @Bean
    public UserDetailsService userDetailService() {
        return username -> userRepository.findByUsername(username)
        .orElseThrow(()-> new UsernameNotFoundException("Usuario no
encontrado!"));
    }
}
```

Corremos nuevamente la aplicación y evidentemente ahora ya está funcionando y está levantando en el puerto 8080. Vamos a evaluar el registro en este caso vamos a hacer la prueba con **Postman**, vamos a crear para eso una nueva petición Http recordemos que el método que debemos trabajar es el método POST, la ruta en nuestro caso va a ser http://localhost:8080 el endpoint que habíamos especificado era /auth/register y tenemos que pasarle en el cuerpo del mensaje los datos. En Body seleccionamos raw y el formato JSON. Enviamos y efectivamente nos devuelve el token.



Vamos a ver si el registro se creó en la base de datos, para eso vamos a abrir MySQL Workbench y vamos a la base de datos securitydb y vamos a la tabla user y efectivamente podemos ver que nuestro usuario ha sido creado, podemos ver también que el password está codificado y que se asigno un rol.



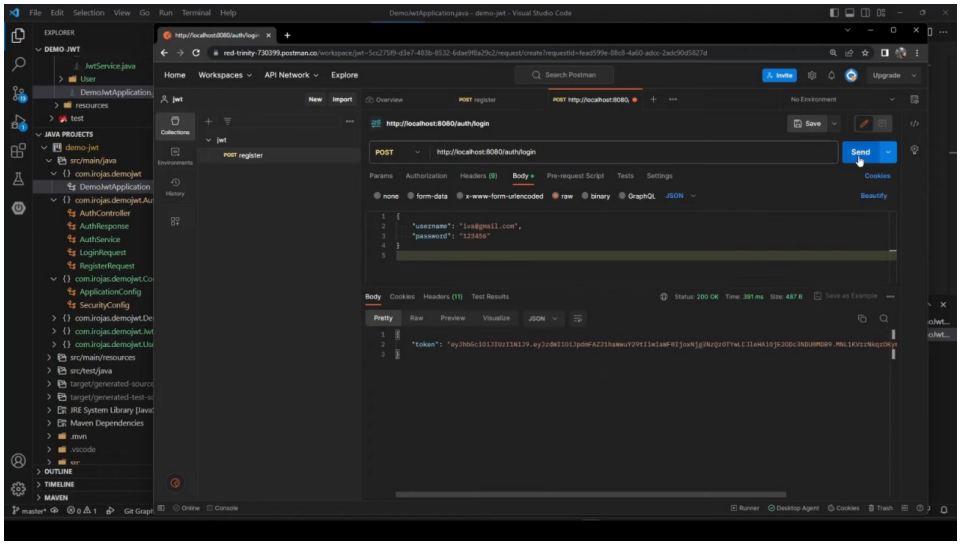
Intentemos ahora con el login, vamos entonces al código, vamos a bajar la aplicación. A continuación **vamos a ir al método login de la clase AuthService** y vamos a implementar el servicio. Lo primero que tenemos que hacer es **crear una instancia del Authentication Manager** porque vamos a necesitar que este usuario se autentique y vamos a **llamar al método authenticate** del **Authentication Manager** que **tiene que recibir por parámetro un objeto del tipo Authentication**. Como no lo tenemos vamos a **crear una nueva instancia de usernamePasswordAuthenticationToken** que **va a recibir por parámetro las credenciales** es decir el username y el password. Ambos datos lo tenemos en nuestro **request**. **Si el usuario se autentico correctamente el siguiente paso va a ser generar el token.** Recordemos que **para generar el token** tenemos que tener **el objeto userDetails**, entonces **lo tenemos que crear**, vamos a **acceder** al mismo **desde el repositorio** llamando al **método findByUsername**. **Si** el mismo **no existe** vamos a **lanzar una excepción**. Ahora si ya tenemos todo para generar el token y **llamar al método getToken**. Le vamos a **pasar por parámetro nuestro userDetails** y finalmente vamos a **retornar AuthResponse**. Recordemos que en la respuesta debemos enviar el token que ya tenemos generado y con esto debería completar nuestro método de autenticación.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Auth;

import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticatio
nToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import com.elavincho.demojwt.Jwt.JwtService;
import com.elavincho.demojwt.User.Role;
import com.elavincho.demojwt.User.User;
import com.elavincho.demojwt.User.UserRepository;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class AuthService {
    private final UserRepository userRepository;
    private final JwtService jwtService;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;

    public AuthResponse login(LoginRequest request) {
        authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(request.getUsername(),
request.getPassword()));
```

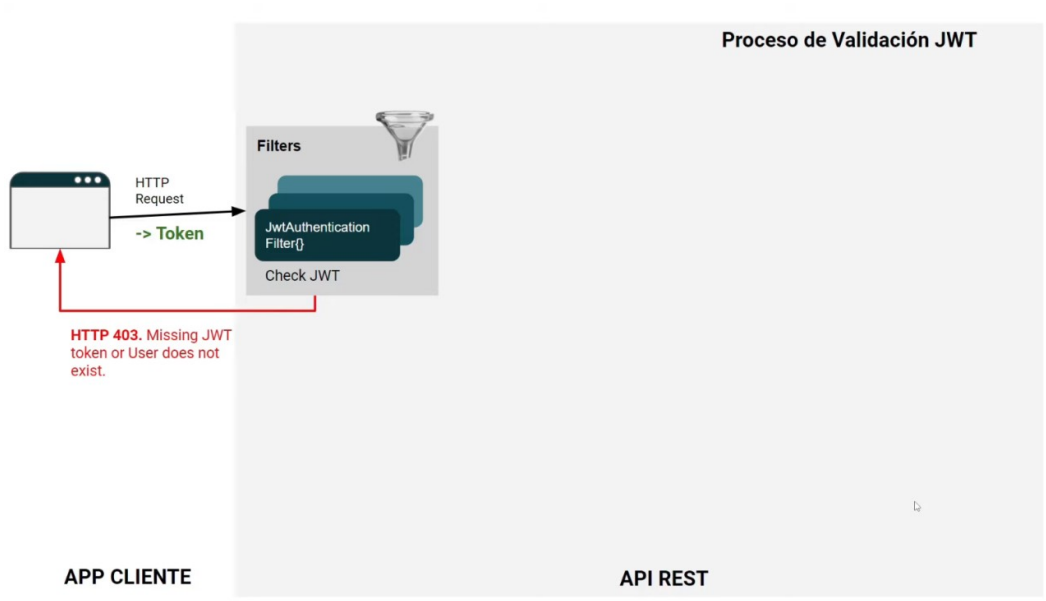
```
        UserDetails user =
userRepository.findByUsername(request.getUsername()).orElseThrow();
        String token = jwtService.getToken(user);
        return AuthResponse.builder()
            .token(token)
            .build();
    }
    public AuthResponse register(RegisterRequest request) {
        User user = User.builder()
            .username(request.getUsername())
            .password(passwordEncoder.encode(request.getPassword()))
            .firstname(request.getFirstname())
            .lastname(request.getLastname())
            .country(request.getCountry())
            .role(Role.USER)
            .build();
        userRepository.save(user);
        return AuthResponse.builder()
            .token(jwtService.getToken(user))
            .build();
    }
}
```

Vamos a ver si funciona, vamos a levantar la aplicación, efectivamente está levantando sin problemas. **Vamos a ir a Postman** y vamos a crear una nueva petición Http, recordemos que el método también es **POST** y la ruta es http://localhost:8080/auth/login. Tenemos que pasarle por parámetro las credenciales del usuario en formato Json. Vamos a enviar.

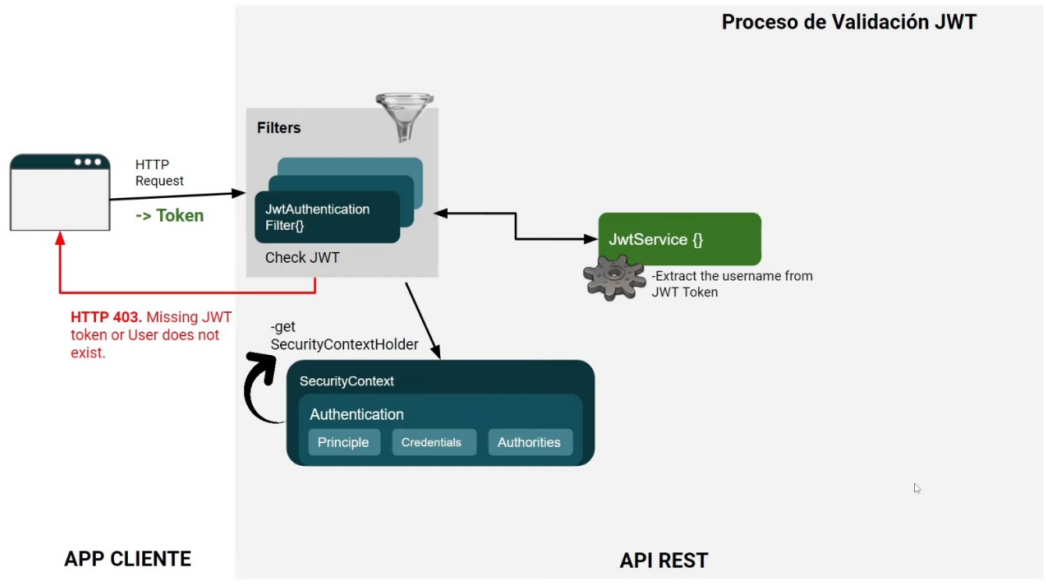


Y vemos que esto funciona efectivamente. Está devolviendo el token. Si modificamos el password **NO** debería devolver el token y debería estar devolviendo un código de estado 403.

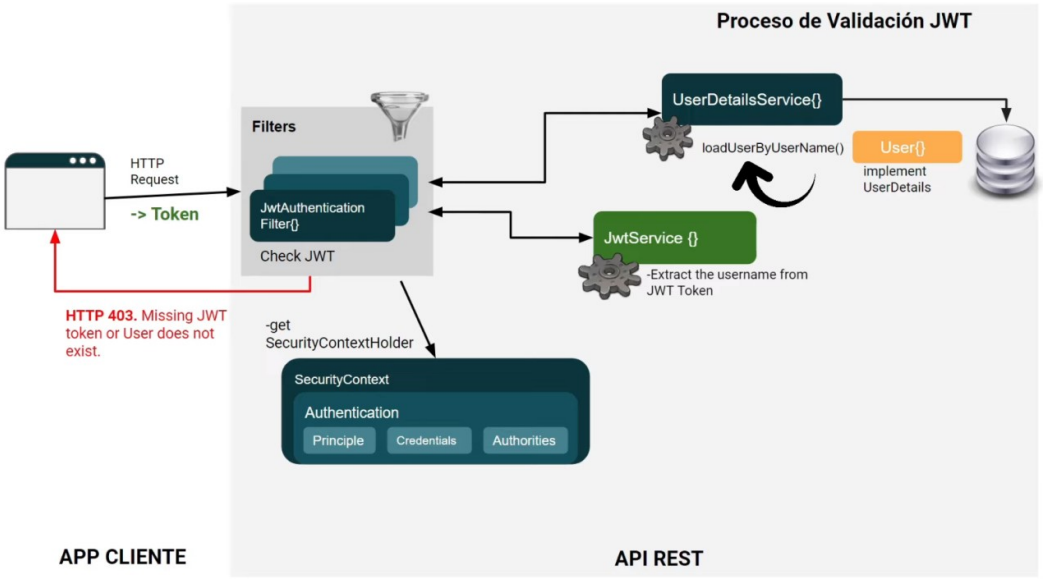
Y como sigue esto, evidentemente el cliente recibió el token y lo va a almacenar localmente, luego lo utilizará para acceder a los endpoint protegidos. Lo va a hacer inyectando el token en la petición http específicamente en el encabezado. Entonces veamos como es el proceso de validación JWT. En este caso se hace una solicitud de nuevo http con la diferencia que ahora tenemos el token en el encabezado de la petición, por supuesto que tenemos el JwtAuthenticationFilter que va a chequear este JWT. Si el mismo no existe o de alguna manera no lo encuentra entonces automáticamente va a devolver un código de estado 403.



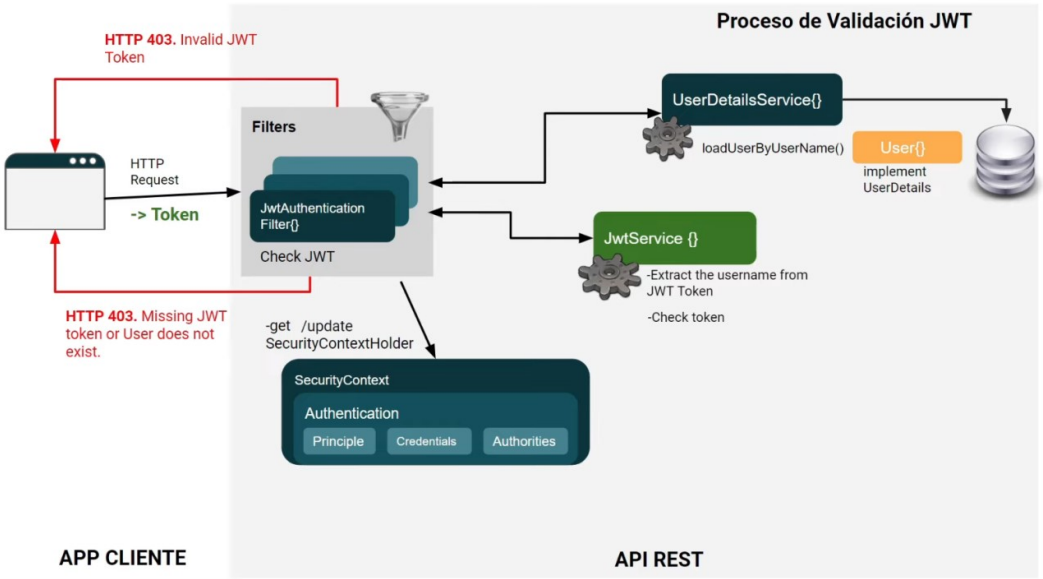
En el caso de que exista va a extraer el username del JWT y va a verificar si lo puede obtener del securityContextHolder.



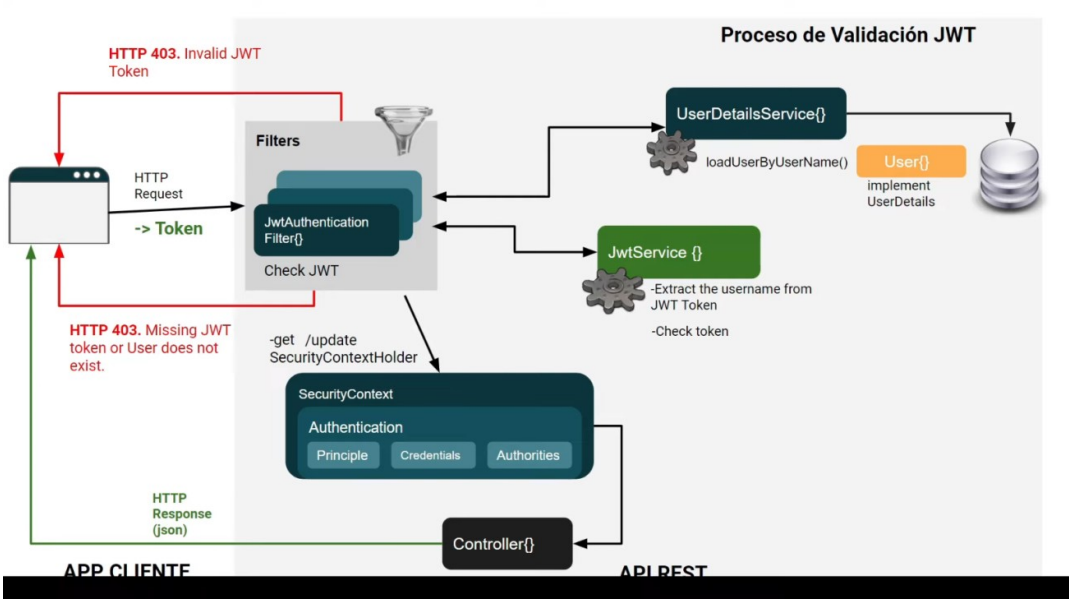
Si no lo consigue en el SecurityContextHolder, entonces va a ir a buscarlo a la base de datos utilizando el userDetailsService y el método loadUserByUserName.



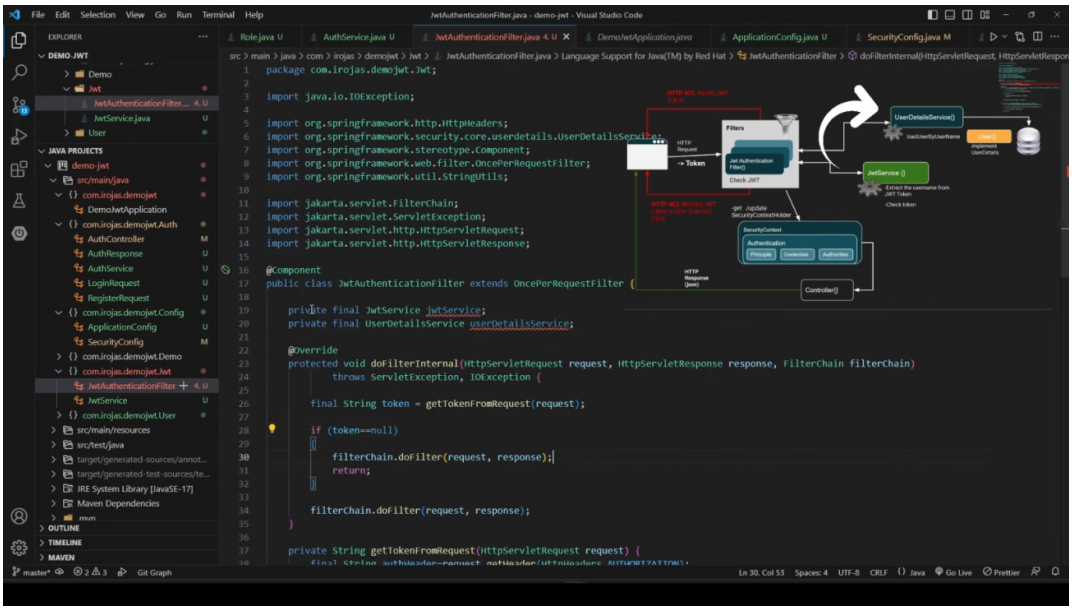
Una vez que lo obtiene va a chequear el token este ok, si la validación falla, entonces también va a devolver un código de estado 403 al cliente.



Caso contrario actualizara el SecurityContextHolder y permitirá el acceso al controlador quien devolverá la respuesta en formato Json u otro formato dependiendo como este configurado al cliente.



Volviendo al código, **vamos a ir a la clase JwtAuthenticationFilter** y en primer lugar **vamos a agregar los servicios**. En este caso vamos a declarar el servicio **JwtService** y el servicio relacionado al **UserDetailsService**.



Finalmente fíjense que en el **método doFilterInternal** donde estábamos obteniendo el token del request también vamos a necesitar al username. Entonces si esto va a todo bien, es decir si el token es distinto de null, entonces tenemos que acceder al username del token, esto nos los va a proveer el servicio JwtService. Y a continuación habíamos dicho si el username es distinto de nulo y que además ese username no lo podíamos encontrar en el securityContextHolder entonces lo íbamos a ir a buscar en la base de datos. Para ello **vamos a crear nuestro objeto UserDetails** y vamos a acceder al mismo gracias al **userDetailsService** y accediendo al **método loadUserByUsername** y le **vamos a pasar por parámetro el username**. Luego vamos a validar si el token era

válido, eso también es algo que nos va a proveer el servicio de JWT **a través de un método** que lo vamos a llamar **isTokenValid**. Por supuesto que en este caso **va a recibir por parámetro** no solo **el token**, sino que **también necesita el userDetails**. Si esto es válido entonces lo que **tenemos** que hacer es **actualizar** el **securityContextHolder**, **lo vamos a hacer creando** un **usernamePasswordAuthenticationToken** y el mismo **va a recibir por parámetro** en primer lugar va a recibir **el userDetails** y **después las credenciales** que nosotros la **vamos a pasar en nulo** y **las Authorities**. Finalmente una vez que creamos usernamePasswordAuthenticationToken lo que **tenemos que** hacer es **setear** el **Details**, para eso **le vamos a pasar una instancia** de **WebAuthenticationDetailsSorce**, vamos a poner que **construya el Details** y **le pasamos por parámetro el request**. Finalmente **vamos a ir al SecurityContextHolder**, vamos a obtener el contexto y **vamos a setear la autenticación**. Con esto ya tenemos configurado nuestro filtro. Vamos entonces a **crear los métodos getUsernameFromToken** y **el método isTokenValid**. Se nos está creando un **error de compilación** de **jwtService** y **userDetailsService**, nos está diciendo que podría estar inicializado, **simplemente** vamos a **agregar la anotation** de Lombok que va a requerir que se inicialicen todos los campos y los atributos en el constructor **@RequiredArgsConstructor**.

Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Jwt;

import java.io.IOException;

import org.springframework.http.HttpHeaders;
import
org.springframework.security.authentication.UsernamePasswordAuthenticatio
nToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.web.authentication.WebAuthenticationDetailsS
ource;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;

@Component
@RequiredArgsConstructor
public class JwtAutenticationFilter extends OncePerRequestFilter {
```

```
    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        final String token = getTokenFromRequest(request);
        final String username;

        if (token==null){
            filterChain.doFilter(request, response);
            return;
        }
        username = jwtService.getUsernameFromToken(token);

        if(username != null &&
SecurityContextHolder.getContext().getAuthentication() == null){
            UserDetails userDetails =
userDetailsService.loadUserByUsername(username);

            if(jwtService.isTokenValid(token, userDetails)){
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                    userDetails,
                    null,
                     userDetails.getAuthorities());

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


SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
    private String getTokenFromRequest(HttpServletRequest request) {
        final String
authHeader=request.getHeader(HttpHeaders.AUTHORIZATION);

        if(StringUtils.hasText(authHeader) &&
authHeader.startsWith("Bearer")){ //Notar que al final de la palabra
"Bearer " hay un espacio
            return authHeader.substring(7);
        }
        return null;
    }
}
```

Ahora **vamos a la clase JwtService** y **buscamos los dos métodos que nos falta configurar**. Lo primero que tenemos que hacer antes de configurar esos dos métodos es **crear un método privado que va a obtener todos los Claims de mi token**. Por supuesto que **va a recibir por parámetro el token**. Vamos a agregar a continuación la **importación** y para acceder a los Claims lo podemos hacer a través d**e la librería de Jwts.** Lo primero que vamos a hacer es **crear un parseBuilder** y a continuación vamos a **especificarle la clave**, en este caso la clave secreta para la firma **setSigningkey(getKey())** y luego **vamos a construir el parse**. También es **importante** que **setear el parseClaimsJwts(token)** para que lo analice y finalmente vamos a obtener el cuerpo. Una vez que tenemos esto **vamos a crear otro método** en este caso va a ser **un método público** y va a ser **un método genérico <T>**. Este método nos va a permitir obtener un Claim en particular. Por supuesto que **va a recibir por parámetro el token y una función**, esta función va especificar el Claims y el tipo de dato genérico y la vamos a llamar **claimsResolver**. Vamos a **agregar** también **la dependencia**, esto es de **java.util** y finalmente **vamos a crear los Claims**. Primero lo **vamos a obtener a todos accediendo al método de Claims** y luego vamos a **aplicar la función** y vamos **retornar el resultado**. Una vez que tenemos esta función ya podríamos obtener el username. **¿Cómo hacemos eso?** Simplemente **vamos a ir a nuestro método getClaim** y le vamos a **pasar por parámetro el token** y **también el Claims en particular**, en este caso es el **getSubjet**. Recordemos que el Subjet es donde vamos a tener alojado el username. Pero para el **isTokenValid** tenemos que tener en cuenta algunas otras cosas como por ejemplo el tema de **la fecha de expiración**. Por lo cual **vamos a crear otro método**, va a **devolver un objeto del tipo Date** y **nos va a devolver la fecha de expiración**. Por supuesto **va a recibir por parámetro el token** y **vamos a utilizar el** mismo **método genérico** como hemos trabajado previamente. En este caso **vamos a acceder a la expiración** una vez que tenemos esto ahora lo que tenemos que hacer es resolver si ese token ha expirado, para ello **vamos a crear otro método** que nos **va a devolver un booleano** y lo vamos a llamar **isTokenExpired**. También **va a recibir por parámetro el token** y **retornamos getExpiration** y vamos a acceder gracias al **método before** pasándole la fecha de este momento por **parámetro**.

Ahora si ya podemos **configurar nuestro método isTokenValid**, para ello lo primero que vamos a hacer es verificar que el username que extraemos o que obtenemos del token corresponde con el que obtenemos de nuestro userDetails, es decir de nuestro objeto. Entonces **vamos a crear una variable username** y vamos a obtener el username de el token **getUsernameFromToken(token).** A continuación vamos a **retornar** si ese username es igual al **userDetails.getUsername** y **además el token no ha expirado**. Si se cumplen ambas condiciones devolverá un true, caso contrario devolverá un false.

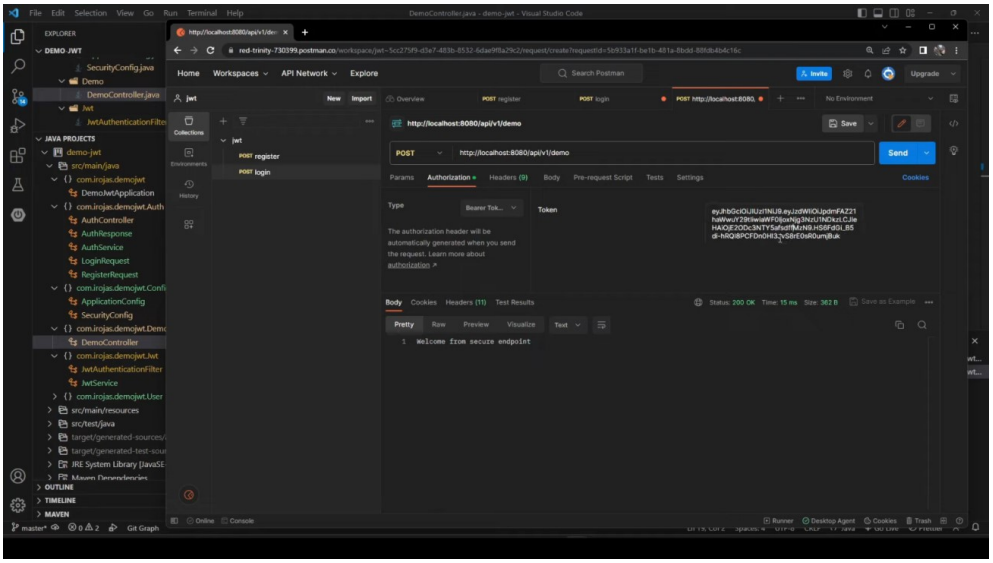
Quedaría de la siguiente manera:

```
package com.elavincho.demojwt.Jwt;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
```

```java
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;

@Service
public class JwtService {

    private static final String SECRET_KEY="123456789";

    public String getToken(UserDetails user) {
        return getToken(new HashMap<>(), user);
    }

    private String getToken(Map<String, Object> extraClaims, UserDetails
user) {
        return Jwts
        .builder()
        .setClaims(extraClaims)
        .setSubject(user.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new
Date(System.currentTimeMillis()+1000*60*60*24))
        .signWith(getKey(), SignatureAlgorithm.HS256)
        .compact();
    }

    private Key getKey() {
        byte[] keyBytes=Decoders.BASE64.decode(SECRET_KEY);

        return Keys.hmacShaKeyFor(keyBytes);
    }

    public String getUsernameFromToken(String token) {
        return getClaim(token, Claims::getSubject);
    }


    public boolean isTokenValid(String token, UserDetails userDetails) {

        final String username = getUsernameFromToken(token);
        return (username.equals(userDetails.getUsername()) &&
!isTokenExpired(token));
    }
```

```
    private Claims getAllClaims(String token){
        return Jwts
            .parserBuilder()
            .setSigningKey(getKey())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }
    public <T> T getClaim(String token, Function<Claims, T>
claimsResolver){
        final Claims claims = getAllClaims(token);
        return claimsResolver.apply(claims);
    }
    private Date getExpiration(String token){
        return getClaim(token, Claims::getExpiration);
    }
    private boolean isTokenExpired(String token){
        return getExpiration(token).before(new Date());
    }
}
```

Volvamos a probar si esto funciona, para eso vamos a levantar nuestra aplicación de Spring Boot y vamos a trabajar nuevamente con **Postman**, vamos primero a ejecutar nuestro login para tener un token actualizado y a continuación vamos a crear una nueva petición http, en este caso la vamos a llamar POST y vamos a llamar al método protegido que habíamos configurado en nuestro DemoControler, este método protegido podemos observar en la ruta /api/v1/demo. Pero al ser un método protegido, tenemos que pasarle el token, volvemos al Body de **Postman** y copiamos el token y lo vamos a pasar en la **pestaña Authorization**, en este caso **vamos a seleccionar Bearer Token**, vamos a pegar nuestro token y finalmente vamos a ejecutar. Como podemos observar nos está devolviendo un código de estado 200, lo cual significa que está perfecto. Podemos acceder incluso al controlador, así que paso por todos los filtros de seguridad en este caso a la autenticación basada en JWT.



Así finaliza el curso de **Como crear el Login con Spring Boot 3 + Spring Security 6 + JWT Authentication.**

https://www.youtube.com/watch?v=nwqQYCM4YT8

https://github.com/elavincho/Spring-Boot-3-Spring-Security-6-JWT-Authentication.git