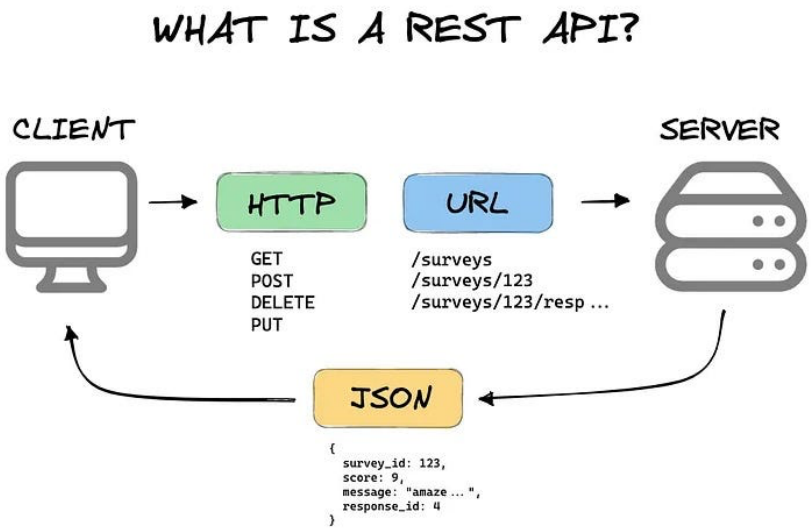
GUIA PRÁCTICA – CREACIÓN DE WS API REST



Por: Ing. Edenilson López

OBJETIVOS:

1. Que el estudiante pueda implementar un servicio web RestFul (API REST) para luego consumirlo desde una aplicación móvil.

INTRODUCCIÓN.

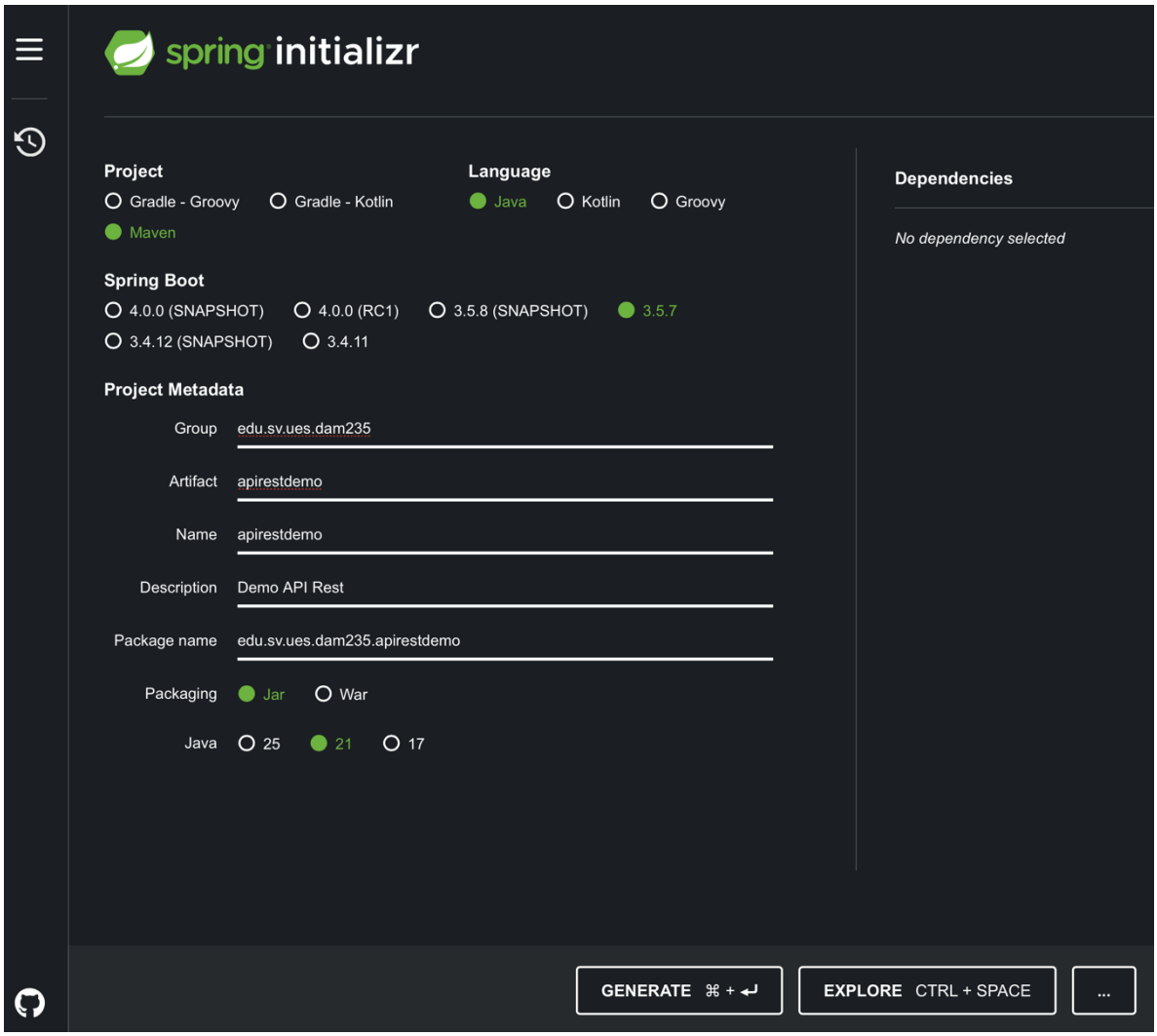
En la presente guía desarrollaremos la práctica relacionada con la creación de un servicio web del tipo API Rest que usaremos para realizar el despliegue de la infraestructura de un Mobile App Server, que posteriormente usaremos conectaremos con una aplicación móvil.
La guía describe los pasos necesarios para generar un proyecto con Spring IO, luego desarrollaremos todo el proyecto con el IDE IntelIJ IDEA, con el cual crearemos toda la estructura del proyecto y las clases necesarias para su implementación, también, se usará una conexión a una base de datos en la nube, una vez construido toda la aplicación de servicio API Rest realizaremos el despliegue desde el IDE y verificaremos su funcionamiento con la herramienta Postman.

Utilidades y dependencias:

IDE: IntelliJ IDEA Community Edition
Postman 11

Paso 1: Crear el proyecto con Spring Initializr.
Ingrese a la dirección URL siguiente:
https://start.spring.io

Y complemente los campos como se muestra a continuación

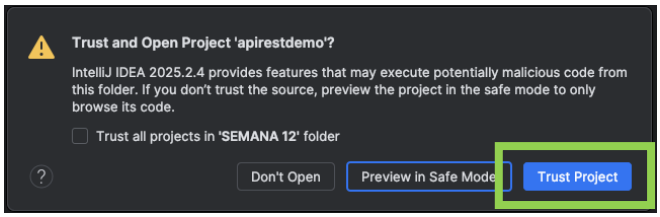


Luego presione el botón **"GENERATE"**

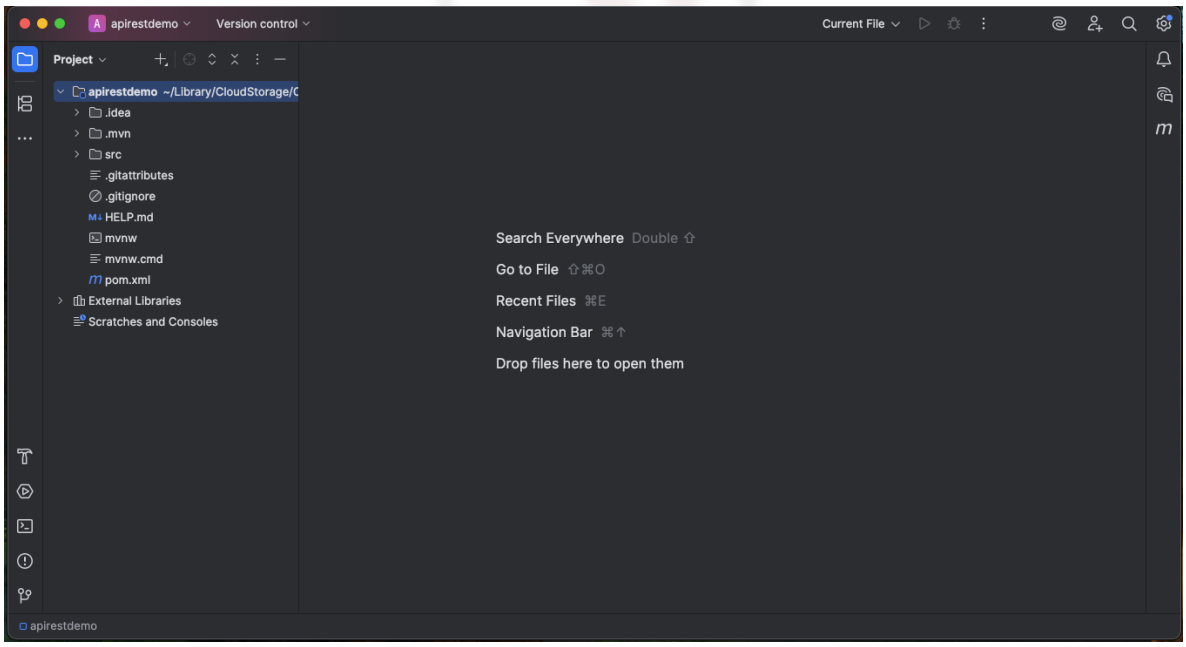
Paso 2: Abrir proyecto con IntelliJ IDE CE.

Una vez descargado e instalado el IDE que usaremos en la presente guía, y teniendo ya descargado el proyecto mover a una carpeta especifica y descomprimir el archivo zip.

En el IDE IntelliJ IDEA, abra la carpera que se generó al descomprimir el archivo zip que se descaro desde Spring.io, si se e muestra un mensaje como el siguiente, de un clic en el botón Trust Project.



Se mostrara una ventana del IDE con el proyecto cargado.



Paso 3: Definición de dependencias (pom.xml)

En el bloque de dependencias encontraremos las siguientes dependencias al archivo pom.xml

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

    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-test</artifactId>
       <scope>test</scope>
    </dependency>
</dependencies>
```

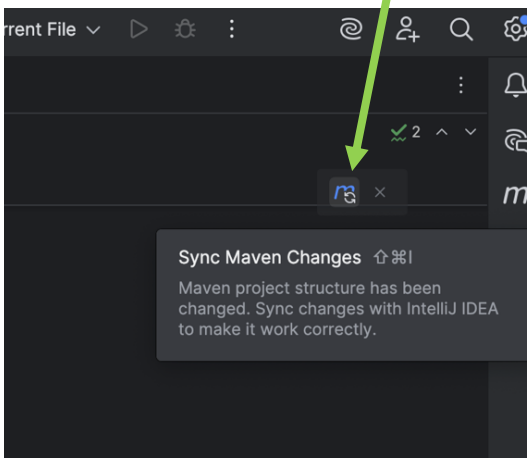
Lo modificaremos para incluir las siguientes dependencias:

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

<!-- spring jpa -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- spring security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
<!-- jakarta servlet -->
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <scope>compile</scope> <!-- Cambiado de provided a compile -->
</dependency>
<!-- JJWT  -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
```

```
</dependency>
<!-- Lombok  -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<!-- Java XML  -->
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
<!-- MariaDB  -->
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <version>3.4.1</version>
</dependency>
```

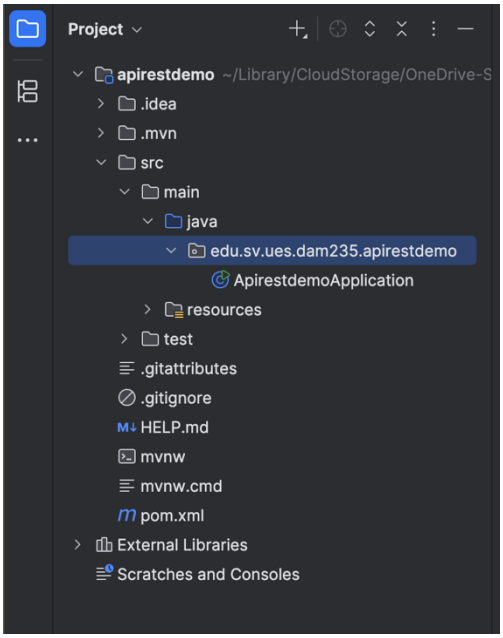
Luego de incluir las dependencias vamos a sincronizar el archivo de dependencias eso lo haremos mediante la opción que se muestra en la imagen siguiente:

Una vez se hallan sincronizado y actualizado las dependencias procederemos a crear las estructuras de paquetes del proyecto



Paso 4: Creación de la estructura de paquetes del proyecto.

La estructura del proyecto sin añadir clases se muestra en la imagen siguiente:

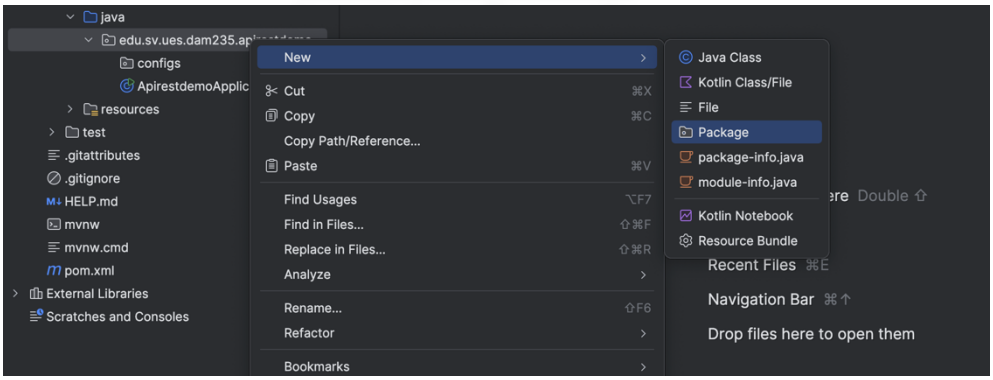


Se crearán los siguientes paquetes dentro de la carpeta:

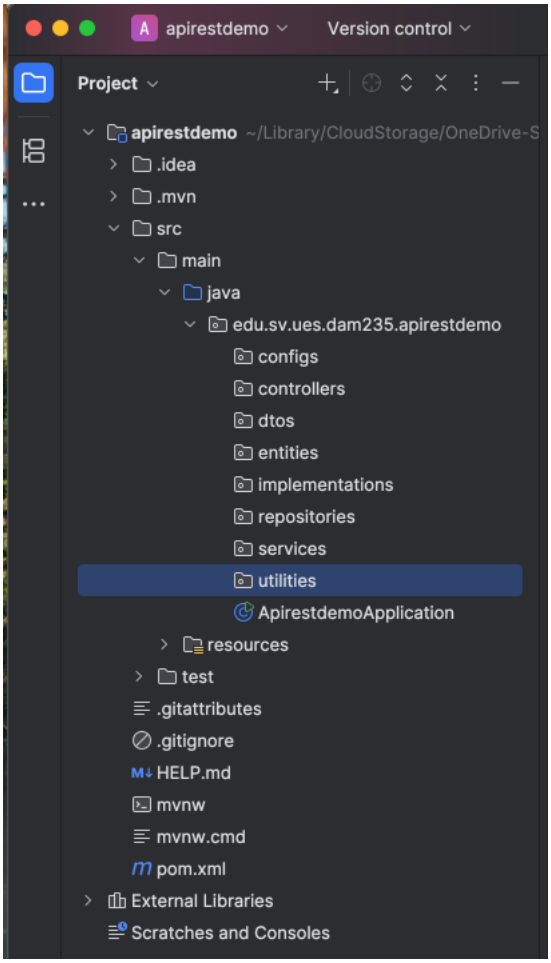
/src/main/java/edu/sv/ues/dam235/apirestdemo/

- configs
- controllers
- dtos
- entities
- repositories
- implementations
- services
- utilities

Para crear los paquetes daremos un clic derecho sobre el paquete raíz, opción New, sub opción Package.



La estructura final del proyecto quedara como la imagen siguiente:



Paso 5: Creación de las clases de entidad.

Dentro del paquete **entities** crearemos la clase java **Usur.java,** la cual se definirá con el siguiente código java.

```
import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "[User]")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private String lastName;
    private String email;
    private String password;
    private Boolean active;
}
```

En el mismo paquete crearemos la clase de entidad **Producto.java**, usando el código siguiente:

```
import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "Product")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "Code")
    private Integer code;

    @Column(name = "Name")
    private String name;

    @Column(name = "Status")
    private boolean status;
}
```

Paso 6: Creación de las clases DTO (Data Transfer Object).

Dentro del **paquete** dto crearemos las clases java necesarias para trasportas los datos entre las capas de la aplicación y para no exponer directamente las entidades de persistencia.

Crearemos la clase LoginDTO para usar vincular los datos provistos por la capa de backend con las propiedades de la clase de identidad User, que creamos en el numeral anterior. El código de la clase LoginDTO.java es el siguiente:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class LoginDTO {
    private String user;
    private String pass;
}
```

Agregaremos un DTO para transferir los datos de los token generados para el manejo de las sesiones en la API Rest. El nombre de la clase será TokenDTO.java, su código es el siguiente:

```
import lombok.Data;

@Data
public class TokenDTO {
    private String token;
    private String expireIn;
    private String msj;
}
```

Finalmente agregaremos un DTO que se usara para enviar los datos desde la capa de bakend cuando se consulte la información almacenada en la entidad Productos. Llamaremos su nombre ProductsDTO.java, su código es el siguiente:

```java
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class ProductsDTO {
    private Integer code;
    private String name;
    private boolean status;
}
```

Paso 7: Creación las clases de Repository de las entidades de persistencia.

Spring Boot usa el patrón de diseño Repository para separar la lógica de acceso a datos de las otras capas de la aplicación, las clases (interfaces) se usan para interactuar con la base de datos de una manera desacoplada.
Por lo anterior, vamos a crear una clase de interface para las entidades de persistencia que hemos creado: User y Product. (se crearán en el paquete repositories)

Nombraremos la clase de repositorio de la clase User con el nombre UserRepository.java, el nombre es una nomenclatura recomendada para no confundir el propósito de las clases de interfaz de tipo Repository. El código de la clase UserRepository.java es el siguiente:

```
import edu.sv.ues.dam235.apirestdemo.entities.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface UserRepository extends JpaRepository<User, Integer> {
    @Query("SELECT u FROM User u WHERE u.email = :email ")
    User findByEmail(@Param("email") String email);
}
```

Como se puede apreciar en el código en esta interfaz se define le método findByEmail() el cual se encarga de recibir un parámetro que será usado por la query SQL que se realizará a la base de datos en la tabla User.

Crearemos adicionalmente la interfaz de la clase Product, la nombraremos ProductsRepository.java, su código es el siguiente:

```
import edu.sv.ues.dam235.apirestdemo.entities.Product;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product,
Integer> {
}
```

Como se podrá observar esta clase no define ningún método, esto se debe a que únicamente se usará para realizar un select a toda la tabla product de la base de datos. Finalmente vale la pena mencionar que ambas interfaces extienden las funcionabilidades de la clase JpaRepository con lo cual se podrán usar o sobre escribir los métodos de esa clase.

Paso 8: Creación de las interfaces de servicio.

Spring Boot usa interfaces de servicio se usan para declarar los métodos de la lógica de negocio que pueden ser implementados en una clase de implementación del servicio la cual es la que se usa para programar la lógica de negocio y el acceso a los repositorios de las entidades de persistencia. En palabras simples la interfaz de servicio define que puede hacer la aplicación y la clase de implementación del servicio define como lo hace.

Dado lo anterior implementaremos dos clases de interfaz de servicio, una llamada AuthServices.java, que servirá para definir las acciones que se pueden realizar en el contexto de la aplicación relacionada con el proceso de autorización o login. Y la clase de interfaz ProductServices.java que definirá las acciones que se pueden realizar en el contexto de la aplicación relacionada con los Productos.

El código de ambas clases de interfaz es el siguiente (se creará en el paquete services):

```
import edu.sv.ues.dam235.apirestdemo.dtos.TokenDTO;

public interface AuthServices {
    public TokenDTO login(String user, String pass);
}
```

```
import edu.sv.ues.dam235.apirestdemo.dtos.ProductsDTO;
import java.util.List;

public interface ProductServices {
    public List<ProductsDTO> getAllProducts();
}
```

Paso 8: Creación de las clases de utilidad para generar el JWT.

Dado que ya tenemos creada la estructura del proyecto necesaria para realizar la implementación de los servicios de la API Rest, es necesario definir la clase de utilidad que se encargara de crear el token JWT que se usara en la implementación del servicio de autorización o login.

Para este proceso definiremos la clase de utilidad JwtUtil.java que es la responsable de implementar las clases propias de las librerías io.jsonwebtoken para generar la estructura del token JWT.

El código de la clase de utilidad es el siguiente (se creará dentro del paquete **utilities**):

```
import edu.sv.ues.dam235.apirestdemo.dtos.TokenDTO;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import
org.springframework.security.authentication.UsernamePasswordAuthenticatio
nToken;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

@Service
public class JwtUtil {

    private String secretKey = "CPqiMDqBnNEXawGoFX7g";

    public String extractUsername(String token) {
        return extractClaims(token, Claims::getSubject);
    }

    public Date extractExpiration(String token) {
        return extractClaims(token, Claims::getExpiration);
    }

    public <T> T extractClaims(String token, Function<Claims, T>
claimsResolver) {
        final Claims claims = extactAllClaims(token);
```

```
        return claimsResolver.apply(claims);
    }

    public Claims extactAllClaims(String token) {
        return
Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token).getBody();
    }

    public boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public TokenDTO generateToken(String userName, UserDetails
userDetails) {
        Map<String, Object> claims = new HashMap();
        String authorities = userDetails.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(","));
        claims.put("authorities", authorities);
        return createToken(claims, userName);

    }

    private TokenDTO createToken(Map<String, Object> claims, String
userName) {

        TokenDTO newToken = new TokenDTO();
        newToken.setExpireIn("1800000");

        String token = Jwts.builder()
                .setClaims(claims)
                .setSubject(userName)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() +
1800000))//30 minutos
                .signWith(SignatureAlgorithm.HS256, secretKey).compact();

        newToken.setToken(token);

        return newToken;
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String userName = extractUsername(token);
        return (userName.equals(userDetails.getUsername()) &&
!isTokenExpired(token));
    }

    public boolean validatedTokenPermission(String token) {
        try {
            String email = extractUsername(token);
            UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(
                    email, null, new ArrayList<>());

SecurityContextHolder.getContext().setAuthentication(authentication);
            return true;
```

```
        } catch (Exception e) {
            return false;
        }

    }


}
```

Al analizar el código de la clase se puede inferir que en esta clase se definen los métodos siguientes:

- createToken()
- validateToken()
- isTokenExpired()
- extractUsername()
- extractExpiration()
- extractClaims()
- extractAllClaims()

Todos estos métodos esta relacionados con la gestión de los tokens JWT y sus reclamaciones (información de campos que componen el token)

Paso 9: Creación de las clases de detalles para la implementación de servicios.

La implementación de Spring Security con JWT requiere la implementación de "clases de detalles de servicios" relacionadas con la seguridad y autenticación de la API Rest. Para que Spring Security pueda autenticar un usuario necesita que una clase de interfaz (que permita sobrescribir métodos de la clase padre UserDetails de Spring Security) devuelva los datos del usuario autenticado (nombre, contraseña, roles, permisos), por lo que, en esta guía crearemos una clase llamada CustomerDetailServices.java para que devuelva los datos del usuario autenticado.

El código de la clase es el siguiente (se creará dentro del paquete **configs**):

```
import edu.sv.ues.dam235.apirestdemo.entities.User;
import edu.sv.ues.dam235.apirestdemo.repositories.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import java.util.ArrayList;

@Service
public class CustomerDetailServices  implements UserDetailsService {

    @Autowired
    private UserRepository userRepo;
    private User userDetail;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        userDetail = userRepo.findByEmail(username);
        if (userDetail != null) {
            try {
                return new
org.springframework.security.core.userdetails.User(userDetail.getEmail(),
                        userDetail.getPassword(),
                        new ArrayList<>());
            } catch (Exception e) {
                return null;
            }

        } else {
            throw new UsernameNotFoundException("Usuario no encontrado");
        }
    }
    public User getUserDetail() {
        return userDetail;
```

```
    }
}
```

Paso 10: Creación de las clases de implementación de servicios.

Como ya se mencionó las clases de implementación de un servicio contienen la lógica de negocio de cómo se implementarán los servicios de la API Rest. Ente apartado realizaremos definición de las clases de implementación de servicios, estas clases son AuthServicesImpl.java que será usado para la implementación del servicio de autenticación o login, y la clase ProductsImpl.java para implementar lo lógica de negocio relacionado con el servicio de productos. El código de ambas clases es el siguiente (se creerán dentro del paquete **implementations**):

AuthServicesImpl.java

```
import edu.sv.ues.dam235.apirestdemo.configs.CustomerDetailServices;
import edu.sv.ues.dam235.apirestdemo.dtos.TokenDTO;
import edu.sv.ues.dam235.apirestdemo.services.AuthServices;
import edu.sv.ues.dam235.apirestdemo.utilities.JwtUtil;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.BadCredentialsException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticatio
nToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

@Slf4j
@Service
public class AuthServicesImpl implements AuthServices {


    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private CustomerDetailServices customerDetailServices;

    @Autowired
    private JwtUtil jwtUtil;


    @Override
    public TokenDTO login(String user, String pass) {
        TokenDTO token = new TokenDTO();
        try {
            Authentication authentication =
```

```
authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(user, pass)
            );

            if (authentication.isAuthenticated()) {
                UserDetails usuarioDetail = (UserDetails)
authentication.getPrincipal();
                if (customerDetailServices.getUserDetail().getActive()) {
                    token = jwtUtil.generateToken(user, usuarioDetail);
                    return token;
                }
            }
        } catch (BadCredentialsException bad) {
            token.setMsj("Credenciales incorrectas!");
            return null;
        } catch (Exception e) {
            log.error("{}", e);
            token.setMsj("Error innesperado");
            return null;
        }
        return null;
    }

}
```

ProductsImpl.java

```
import edu.sv.ues.dam235.apirestdemo.dtos.ProductsDTO;
import edu.sv.ues.dam235.apirestdemo.entities.Product;
import edu.sv.ues.dam235.apirestdemo.repositories.ProductRepository;
import edu.sv.ues.dam235.apirestdemo.services.ProductServices;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Slf4j
@Service
public class ProductsImpl implements ProductServices {
    private final ProductRepository productRepository;

    private ProductsImpl(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @Override
    public List<ProductsDTO> getAllProducts() {
        List<ProductsDTO> result = new ArrayList<>();
        List<Product> items = this.productRepository.findAll();
        for (Product item : items) {
            result.add(new ProductsDTO(item.getCode(), item.getName(),
item.isStatus()));
        }
        return result;
```

```
    }
}
```

Paso 11: Creación de las clases controladores.

Ya estamos casi por finalizar la creación y despliegue de la API Rest, pero nos falta crear unas clases que representan los puntos de entrada a las peticiones HTTP de la API Rest, estas son clases "Controller" o controladores. Estas clases son las encargadas de atender las solicitudes de los clientes (dispositivo móvil, navegador, postman etc.). para nuestro caso práctico crearemos dos clases de tipo controller: AuthController.java y ProductController.java las cuales expondrán los endpoints **/auth/login** y **/products** respectivamente. El código de ambas clases es el siguiente (se crearán en el paquete **controllers**):

AuthController.java

```
import edu.sv.ues.dam235.apirestdemo.dtos.LoginDTO;
import edu.sv.ues.dam235.apirestdemo.dtos.TokenDTO;
import edu.sv.ues.dam235.apirestdemo.services.AuthServices;
import lombok.extern.slf4j.Slf4j;
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;

@Slf4j
@RestController
@RequestMapping("/auth")
public class AuthController {

    final private AuthServices authServices;

    private AuthController(AuthServices authServices) {
        this.authServices = authServices;

    }

    @PostMapping("/login")
    public ResponseEntity<TokenDTO> login(@RequestBody LoginDTO
authRequest) {
        try {
            System.out.println("DTO enviado : " +
authRequest.toString());
            TokenDTO token = authServices.login(authRequest.getUser(),
authRequest.getPass());
            if (token == null) {
                return ResponseEntity.status(401).build();
            } else {
                return ResponseEntity.ok(token);
            }
```

```
        } catch (Exception e) {
            log.error("{}", e);
        }
        return null;
    }
}
```

ProductController.java

```
import edu.sv.ues.dam235.apirestdemo.dtos.ProductsDTO;
import edu.sv.ues.dam235.apirestdemo.services.ProductServices;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Slf4j
@RestController
@RequestMapping("/products")
public class ProductController {

    final private ProductServices productServices;

    private ProductController(ProductServices productServices) {
        this.productServices = productServices;
    }

    @GetMapping
    public ResponseEntity<List<ProductsDTO>> getAllItems() {
        try {
            List<ProductsDTO> items = productServices.getAllProducts();
            if (items.isEmpty()) {
                return ResponseEntity.status(204).build();
            } else {
                return ResponseEntity.ok(items);
            }
        } catch (Exception e) {
            log.error("{}", e);
        }
        return null;
    }

}
```

Paso 12: Creación de las clases que implementan la seguridad basada en JWT.

Para implementar la seguridad de una API Rest echa con Spring Boot se necesitan dos clases fundamentales, estas son JwtFilter y SecurityConfig.
Ambas clases trabajan en conjunto para implementar la seguridad del acceso a las endpoints de la API Rest. Por un lado, la clase JwtFilter implementa el filtro de autenticación, es decir, intercepta todas las peticiones HTTP que llegan a la API y verifica que se incluya un token valido. Si el token es válido, se le permite al usuario acceder al contexto solicitado en la petición HTTP. Por otro lado, la clase SecurityConfig define que rutas están protegidas y cuales son públicas, que filtros usar y que mecanismo de autenticación emplear (para esta guía JWT).

Dada la explicación anterior, crearemos las dos clases mencionadas dentro del paquete **configs**, el código de ambas clases es el siguiente:

JwtFilter.java

```
import edu.sv.ues.dam235.apirestdemo.utilities.JwtUtil;
import io.jsonwebtoken.Claims;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Configuration
@EnableWebSecurity
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;


    Claims claims = null;


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

```
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "*");
        response.setHeader("Access-Control-Allow-Headers", "*");
        response.setHeader("Access-Control-Allow-Credentials", "false");
        response.setHeader("Access-Control-Max-Age", "3600");
        System.out.println("********** CORS Configuration Completed
**********");

        // Verificar si es una solicitud preflight (OPTIONS)
        if ("OPTIONS".equalsIgnoreCase(request.getMethod())) {
            // Responder con HTTP 200 y no continuar con el filtro
            response.setStatus(HttpServletResponse.SC_OK);
            return;
        }

        String path = request.getServletPath();
        if (path.startsWith("/auth/login")
                || path.startsWith("/auth/verify-token")
                || path.startsWith("/swagger-ui/")
                || path.startsWith("/v3/")) {
            filterChain.doFilter(request, response);
            return;
        }

        String authorizationHeader = request.getHeader("Authorization");
        String token = null;

        if (authorizationHeader != null &&
authorizationHeader.startsWith("Bearer ")) {
            token = authorizationHeader.substring(7);
        } else {
            token = authorizationHeader;
        }


        if (jwtUtil.validatedTokenPermission(token)) {
            filterChain.doFilter(request, response);
        } else {
            // Token no válido o no proporcionado: enviar error 401
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            response.getWriter().write("No autorizado: Token no es el
correcto o no proporcionado");
        }

    }

    public Boolean isAdmin() {
        return "admin".equalsIgnoreCase((String) claims.get("role"));
    }

    public Boolean isOther() {
        return "user".equalsIgnoreCase((String) claims.get("user"));
    }

}
```

SecurityConfig.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configurati
on.AuthenticationConfiguration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;
import
org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthentic
ationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private JwtFilter jwtFilter;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return
PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    protected SecurityFilterChain securityFilterChain(HttpSecurity
httpSecurity) throws Exception {

        httpSecurity
                .csrf(csrf -> csrf.disable())
                .cors(cors -> cors.disable())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/auth/login",
                                "/swagger-ui/*", "/v3/*",
                                "/v3/api-docs/swagger-config")
                        .permitAll()
                        .anyRequest().authenticated() // Requiere
autenticación para las demás rutas
```

```
                );
        httpSecurity.addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);
        return httpSecurity.build();

    }

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

}
```

Paso 13: Configuración de application.properties.

Como último paso antes de la compilación y prueba de la API Rest, vamos a configurar el archivo **application.properties** el cual contiene los parámetros generales de la API este archivo se encuentra en la carpeta **resources**, su contenido original es:

```
spring.application.name=apirestdemo
```

Agregaremos las siguientes líneas de configuración:

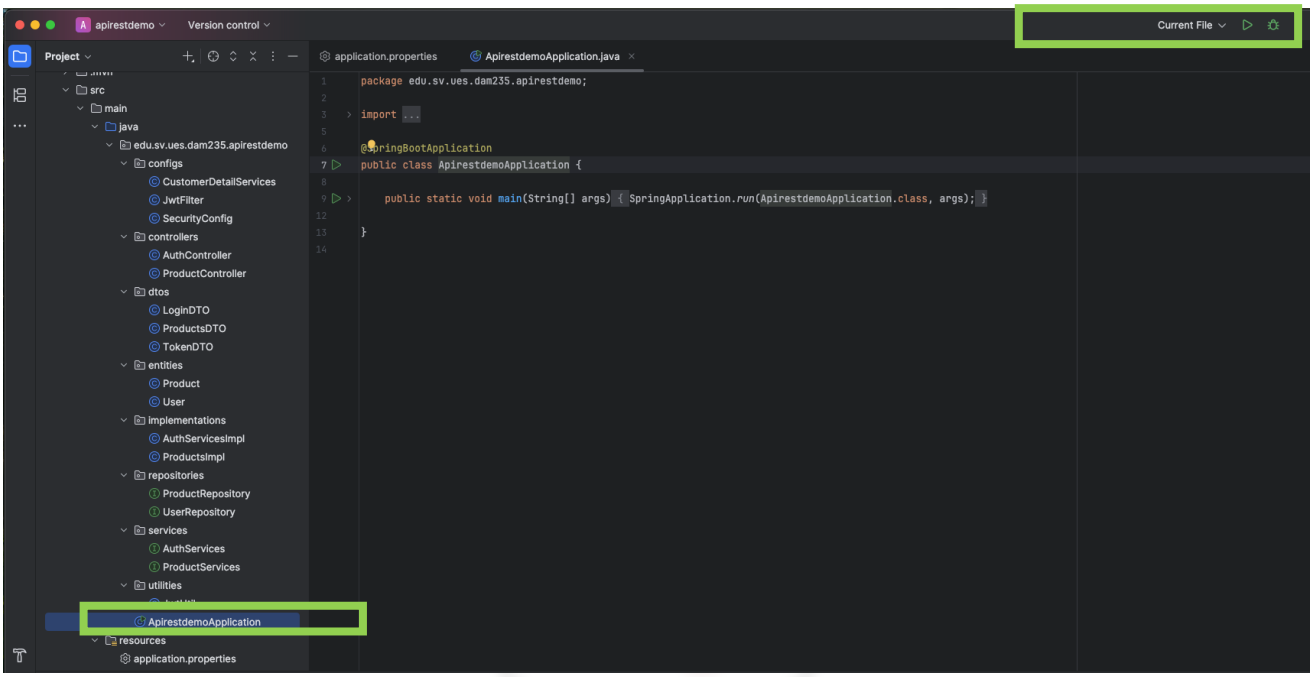
```
server.port=8085
server.servlet.context-path=/api/v1/demoapirestdam235
app.version=1.0.0

## Configuración de la conexión a MariaDB
spring.datasource.url=jdbc:mariadb://52.7.114.182:3306/DBTemp
spring.datasource.username=dam235
spring.datasource.password=D@m235U35._
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver

# Configuración de Hibernate/JPA
spring.jpa.database-platform=org.hibernate.dialect.MariaDBDialect
spring.jpa.properties.hibernate.globally_quoted_identifiers=false
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# configuracion de logs
logging.level.root=INFO
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

En este archivo se define el puerto donde se desplegará la API Rest, la ruta del contexto a usar, la configuración a la base de datos a la cual nos conectaremos, que para el caso es una base de datos MariaDB de nombre DBTemp, que esta hospedada en el host con IP 52.7.114.182, también se configuran el usuario y la clave y el driver JDBC de conexión a la DB. Opcional mete se habilitan parámetros de JPA para lo relacionado a la persistencia de la base de datos y el dialecto usado por la base de datos.

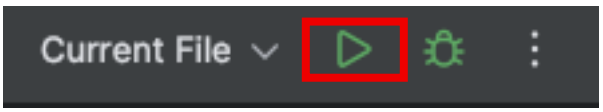
Finalmente se definen parámetros de comportamiento relacionados con los logs que generará la API Rest. Ahora ya estamos listos para compilar y ejecutar la API Rest.

Paso 14: Compilación y ejecución de la API Rest.

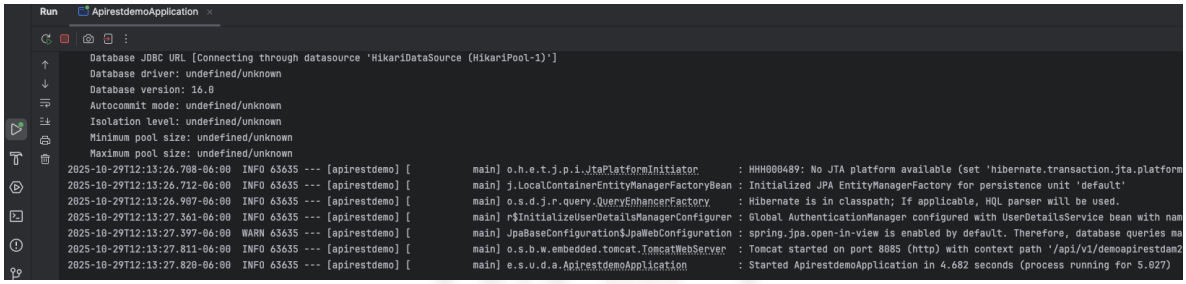
En el Ide ubicaremos la clase ApirestdemoApplication tal como se muestra en la imagen siguiente:



En la esquina superior derecha encontraremos los botones para ejecutar la Aplicación, daremos clic en el botón play.



Si la aplicación se ejecuta sin error observaremos una salida en la ventana de ejecución como la siguiente:



Y la barra de ejecución cambiara al de la siguiente manera

La aplicación API Rest está en ecuación uy estamos listos para probarla, usaremos Postman para esta actividad.

Paso 15: Pruebas la API Rest.

Una vez la API Rest este en ejecución, abriremos Postman para realizar las peticiones HTTP a los endpoint de la API.

La forma de realizar las solicitudes http en Postman es la siguiente

<protocolo><host:puerto><context.path><endpoint>

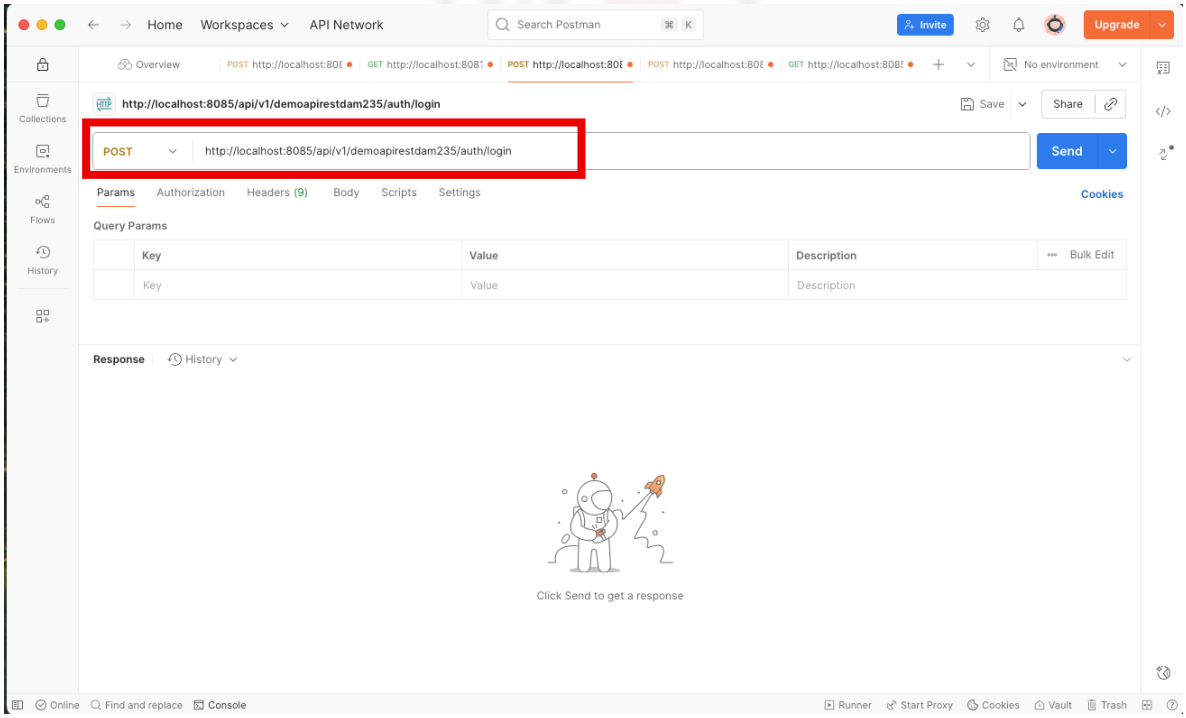
Po lo que usaremos la siguiente URL para realizar las pruebas en Postman para el endpoint **/login**

http://localhost:8085/api/v1/demoapirestdam235/auth/login

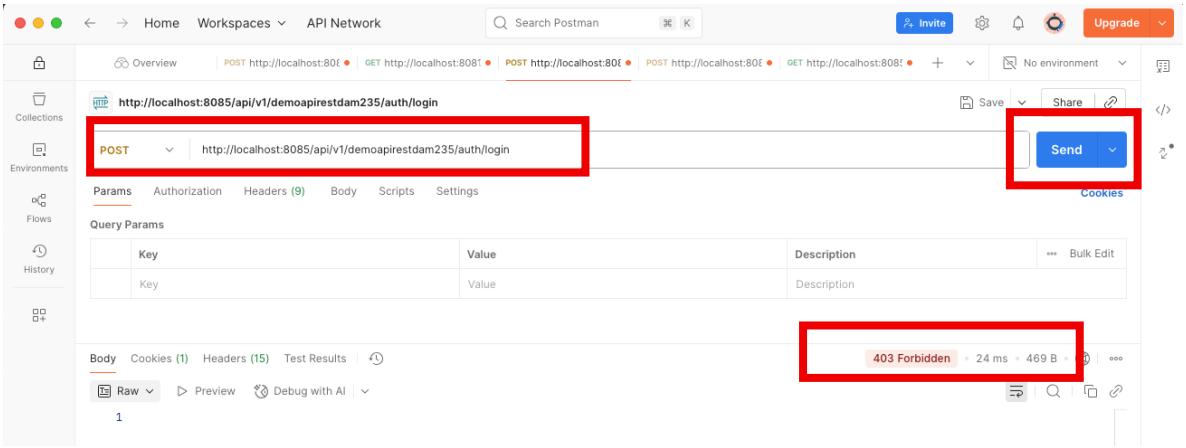
En la base de de ejemplo ya se encuentra creado un registro de usuario con los siguientes atributos:

User: mmartinez@temp.com
Pass: Temp123.

Abrimos Postman y pegamos la URL del primer endpoint.



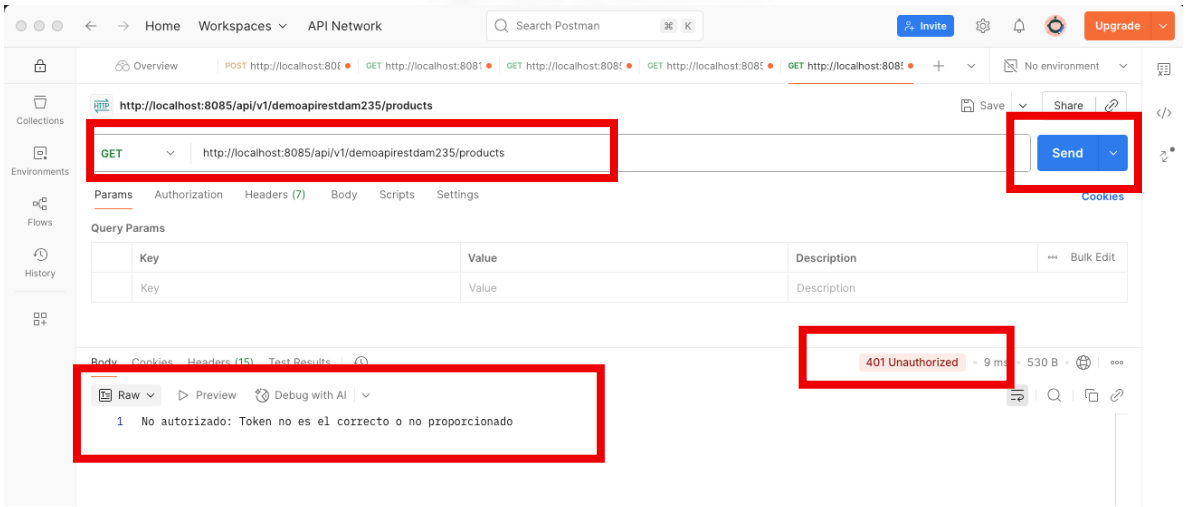
Si ejecutamos postan solo con el endpoint sin pasar los parámetros de inicio de sesión obtendremos el mensaje siguiente:



La API responde con el código http 403 Forbidden (No permitido)

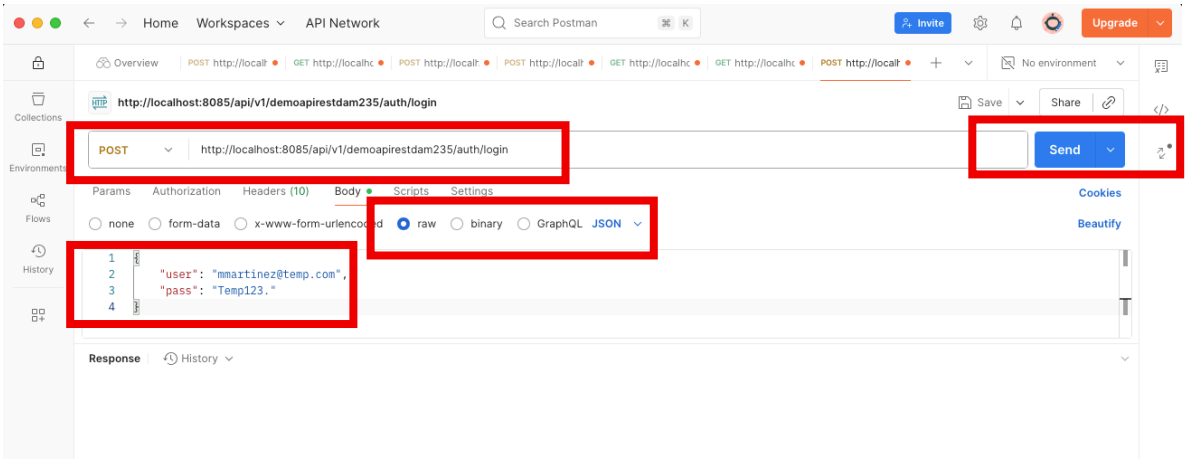
Probaremos el segundo endpoint

http://localhost:8085/api/v1/demoapirestdam235/products

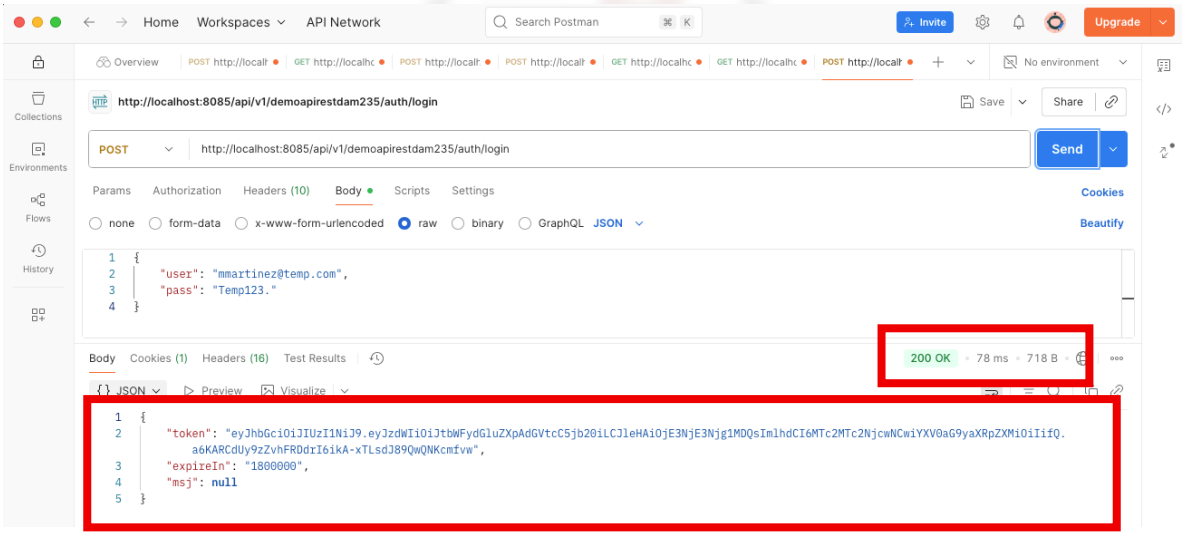


La API responde con el código http 401 (Sin autorización)

Ahora enviaremos los parámetros de usuario y la calve en el body de la petición http en formato raw json:

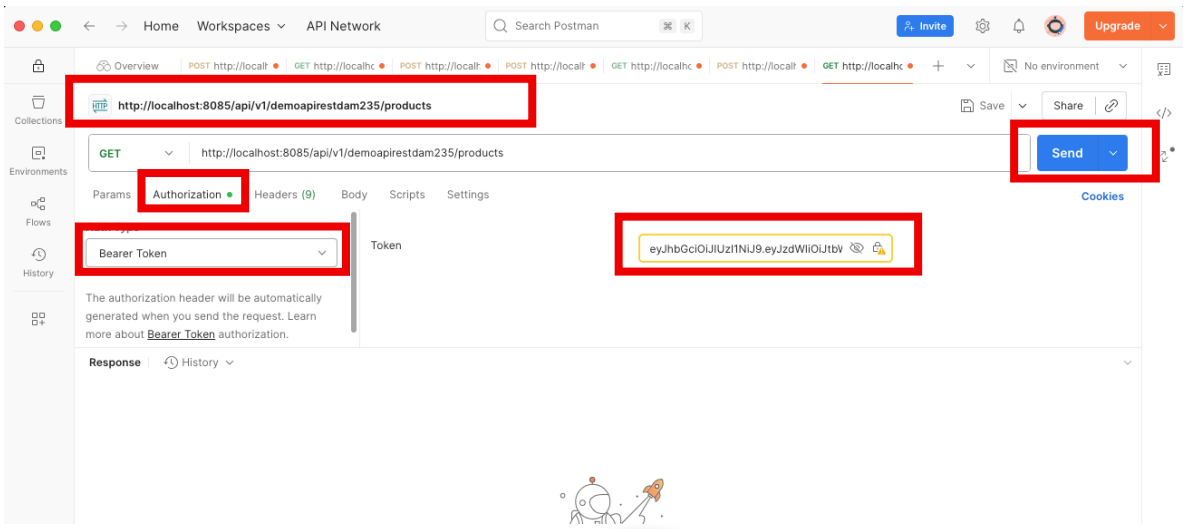


Tras esa petición el resultado es el siguiente:

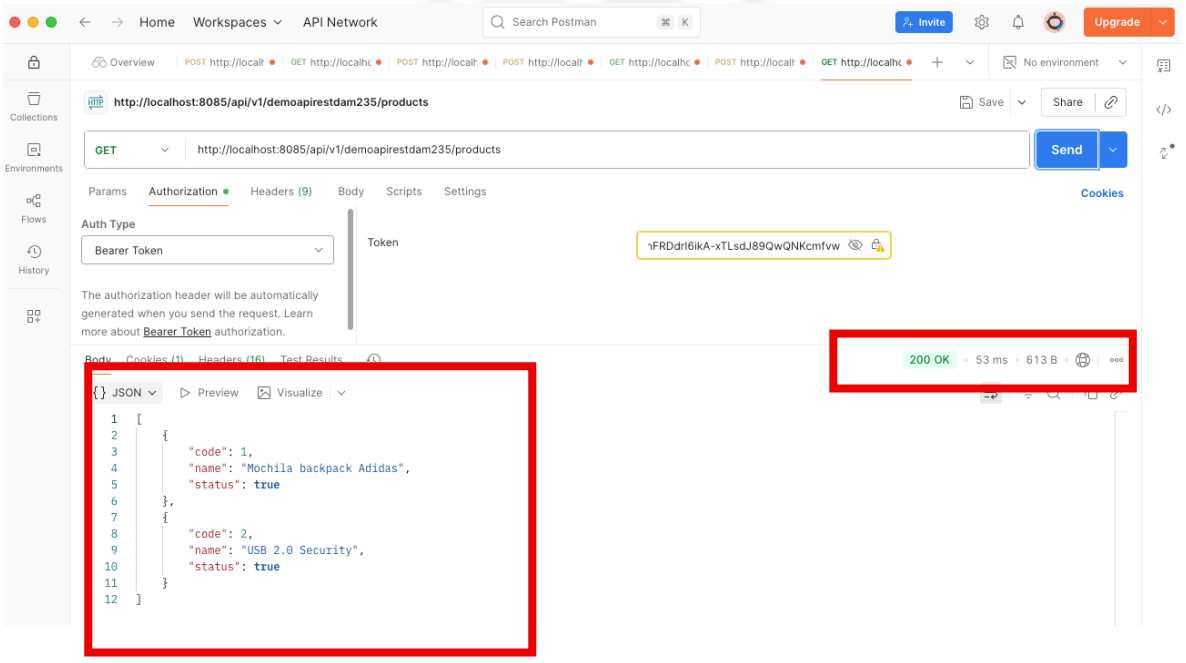


Como podemos observar el usuario ya se ha autenticado, y se le ha generado un Token JWT.

Ahora probaremos el segundo endpoint enviando como parámetro el token que se nos proporcionó a la hora de autenticarnos.



El resultado es el siguiente:



Y de esa manera hemos comprobado que nuestra API Resta está funcionando.

Actividades a desarrollar (Tarea Opcional).

1. Una vez que la API Rest sea funcional se solicita elaborar un proyecto en Android Estudio en el cual se implementaran las dependencias Retrofit y Gson para conectar una activity de login con la API Rest y que luego de haber validado el usuario y password se muestre otra Activity en donde se pueda consultar los productos de la base de datos mediante el consumo el endpoint /productos
2. Desarrollar el endpoint necesario para realizar el logout de la API.
3. Desarrollar los endpoint para realizar la creación de nuevas cuentas de usuario.
4. Desarrollar los endpoint para realizar las operaciones CRUD de persistencia de la entidad Productos.

Indicaciones de la actividad.

Se habilitará un espacio en la semana 15 para subir un archivo en formato ZIP que incluya el proyecto y un video demostrativo del funcionamiento.

Conclusiones.

Es importante observar que la guía pretende ser una herramienta demostrativa del procedimiento técnico esencial para la creación de una API Rest con Spring Boot y Spring Security.

Existen consideraciones de seguridad que no ha sido incluidas para mantener la simplicidad de la guía, por lo que no se recomienda replicar este desarrollo en un entorno de producción sin antes contemplar los elementos tecnológicos, de desarrollo y de infraestructura mínimos y necesarios para garantizar la seguridad de la información

ANEXOS.

ULR Github

https://github.com/edenilsonsv/apirestdemo

Script SQL.

```
CREATE DATABASE IF NOT EXISTS DBTmp;

USE DBTmp;

CREATE TABLE user (
  id int(11) NOT NULL,
  name varchar(256) CHARACTER SET utf8mb3 COLLATE utf8mb3_spanish_ci NOT
NULL,
  last_name varchar(256) CHARACTER SET utf8mb3 COLLATE utf8mb3_spanish_ci
NOT NULL,
  email varchar(128) CHARACTER SET utf8mb3 COLLATE utf8mb3_spanish_ci NOT
NULL,
  password varchar(256) NOT NULL,
  active bit(1) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;


CREATE TABLE product(
  code int(11) NOT NULL AUTO_INCREMENT,
  name varchar(255) DEFAULT NULL,
  status bit(1) DEFAULT NULL,
  PRIMARY KEY (code)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;


INSERT INTO user VALUES
(1,'Marcos','Martinez','mmartinez@temp.com','{bcrypt}$2a$10$uvWNfki.WxQhUD6e
51di0uvC9G0DiGly7/PvyLR0Ay0mpFV24YklG',1);
INSERT INTO product VALUES (1,'Mochila backpack Adidas',1),(2,'USB 2.0 Security',1);
```