MUSIMASTER

Hecho por: Rocío Palao, Mohamed Asidah, Daniel Carmona, Jeremy Ramos, Alejandro López

Objetivo del proyecto

Nos han contratado con la tarea de realizar una aplicación de gestión sobre una página de venta de guitarras y bajos

r

Índice

Diseño y propuesta de solución establecidas

Diseño

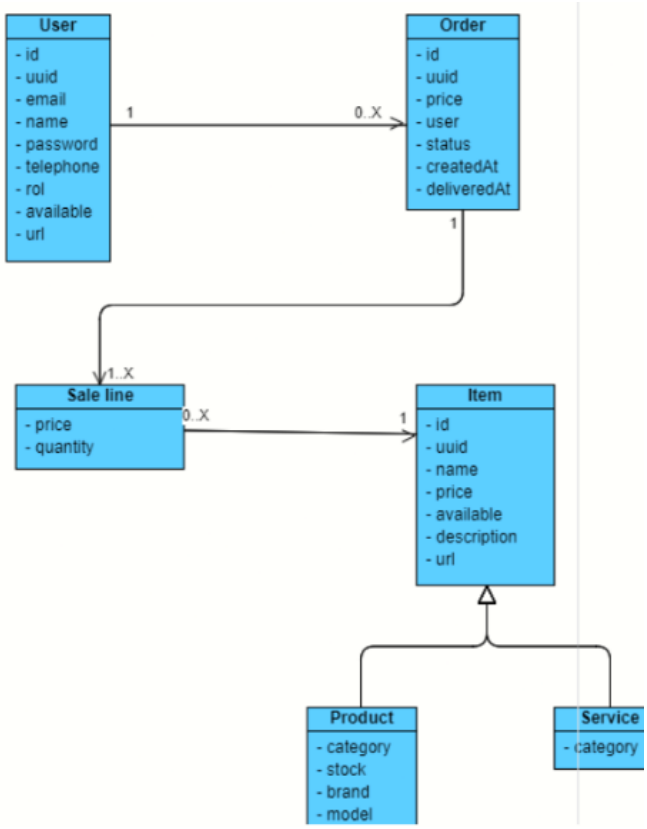
Nos hemos centrado en una estructura de código mediante contenedores donde almacenamos las clases de nuestro programa de tal forma que se vea sobre el mismo la idea principal de nuestro proyecto: La Gestión de MicroServicios sobre un mismo programa.

Propuesta de solución

Nos hemos planteado una solución afable y correlacionada a nuestros conocimientos y utilizando tecnologías varias las cuales veremos posteriormente en el programa. Como base tendremos la idea de integrar distintas tecnologías para la gestión de la información en este caso distintos sistemas de gestión de base de datos(Relacionales y NoSQL) las cuales redactaremos más adelante. También la forma de gestionar los mismos serán a través de Frameworks tales Ktor, Spring los cuales serán explicados a lo largo del documento. Referente a esto utilizaremos una arquitectura de microservicios la cual se desenvolverá en 4 partes donde una parte estará montada toda la lógica de la seguridad la cual se manejará mediante el sistema de tokens y SpringSecurity. La maquetación será: 4 Api´s donde: Productos y Servicios(Actúan con Spring + MariaDB), Usuarios(Spring + SpringSecurity), Pedidos(Actúa con Ktor y Mongo) + ApiGeneral (Gestionada con Ktor). El porqué de esta arquitectura será detallado posteriormente en el documento

Diagrama de Clases.

Hemos repartido el problema en ciertas clases las cuales disponen de sus atributos y el tratamiento de las distintas llamadas y el detalle de observar los datos que están pasados por referencias o embebidos sobre otro documento etc.



Explicación del Diagrama de Clases

La clase Usuario la cual se compone de los atributos básicos de entrada de cualquier usuario estándar y nos ofrece introducir el campo rol para diferenciarlos entre sí porque dependiendo del tipo de usuario tendrán un menú y/o operaciones distintas, Junto con ciertos permisos para ejecutar dichos métodos. Existe una relación entre usuario y pedido la cual se representa al guardar el id del mismo sobre el pedido y que al actualizar un usuario los campos que serán actualizados serán pocos ya que para deshabilitar el mismo tenemos un campo avaliable el cual nos dice si está disponible el usuario, y pedido también cuenta con campos que no cambian al actualizar solo se cambiará el precio y su status para comprobar el estado del mismo .

Por consiguiente, la clase pedido también tiene una relación con la clase con línea de venta que serán donde se almacene la lógica del precio total del pedido.

Dicha línea de venta se compone a su vez de ítems los cuales tienen campos descriptivos y junto a la lógica de la aplicación un campo disponible para solo deshabilitar el estado de este. Dentro de ítems podemos ofrecer Productos y Servicios los cuales tendrán campos descriptivos tales como el nombre del mismo y/o la categoría.

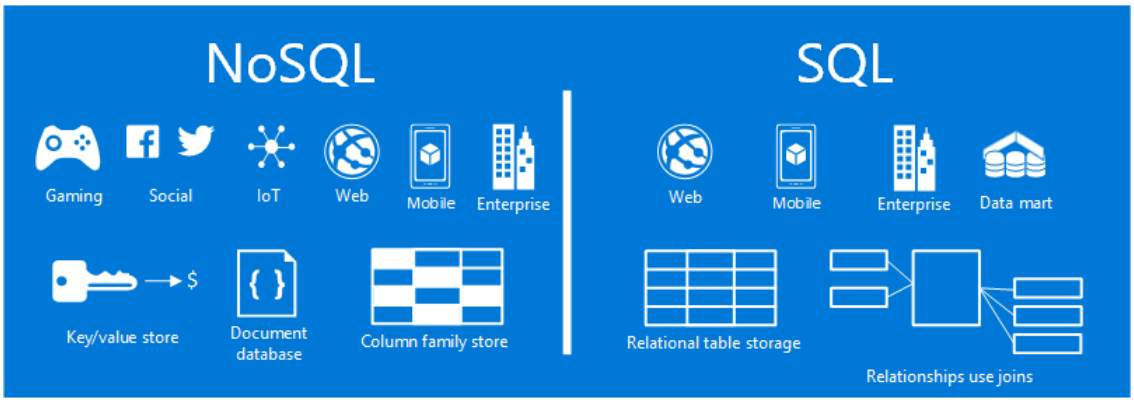
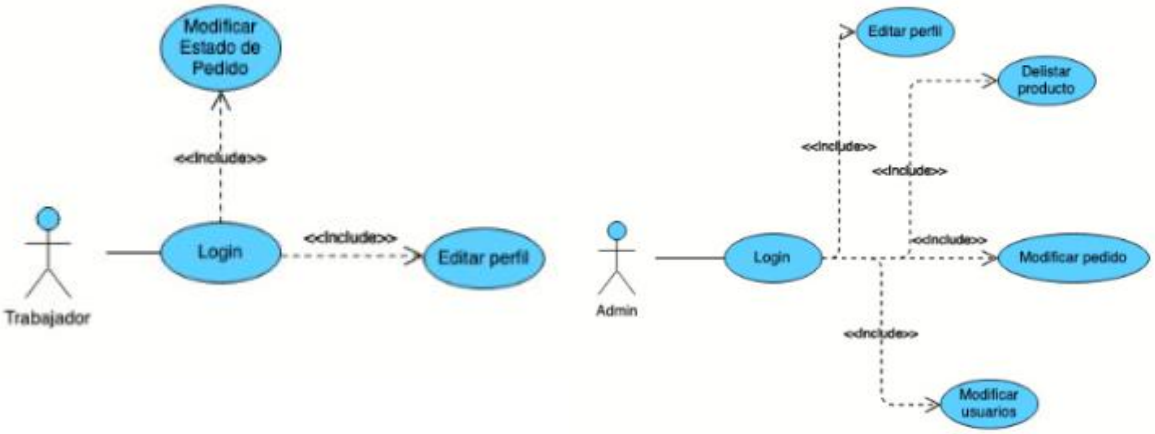


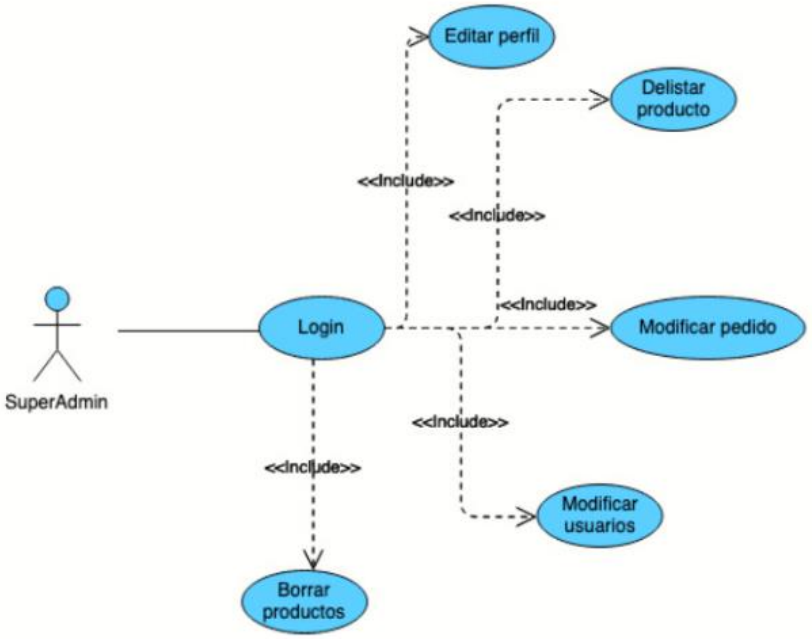
Diagrama de Casos de Uso

Trabajador

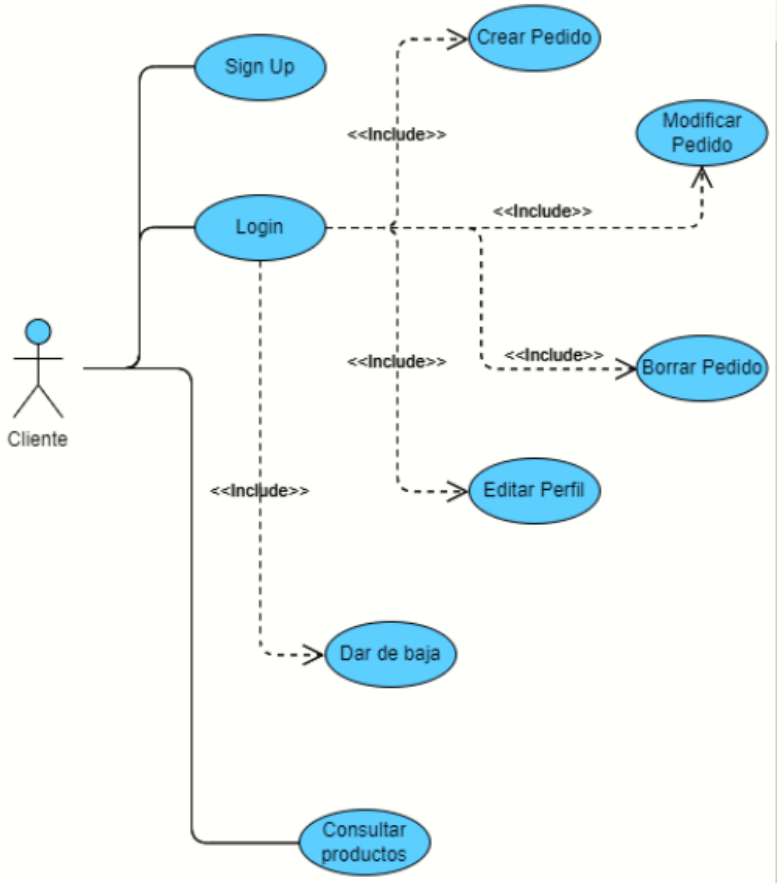
Admin



SuperAdmin



Cliente



Arquitectura del sistema

Ktor + Mongo

Utilizamos ktor gracias a que nos proporciona un sistema de programación asíncrono gracias a las corrutinas lo que nos permite una gestión eficiente de los recursos. Nos ofrece la posibilidad de una arquitectura basada en **plugins** y de módulos y/o librerías de forma eficiente y rápida. También nos ofrece el manejo de solicitudes http lo cual nos hará más amena la parte de Pedidos y ApiGeneral integrada en este programa.
Utilizamos Mongo en este apartado ya que nos ofrece ventajas frente a una base de datos Relacional que nos permitirán manejar de forma íntegra gran cantidad de datos junto con su diseño NoSQL

Spring + Security

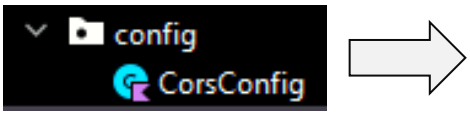
Utilizamos Spring mas especifico SpringBoot para el manejo de las Api´s Productos, Servicios y Usuarios. Lo decidimos de esta manera gracias a que podremos crear Api's con facilidad ya que cuenta con un conjunto de características RESTful las cuales nos ayudarán con la gestión de solicitudes http como ktor y como apunte importante es su gran escalabilidad y su buena integración frente a otros Frameworks como en este caso que hemos integrado la comunicación entre Ktor y Spring. La seguridad está basada en Spring 3 ya que es una versión reciente con lo cual estará actualizada y sobre todo la seguridad con spring es muy completa contando con clases específicas para la gestión de usuarios tales como: UserDetails y UserDetailService. Gracias a esta la gestión de usuarios se ve monitorizada por spring y en todo caso podremos configurar nosotros la gestión de estos en caso beneficiario.

Disposición de las Apis de forma Programática

Productos/Servicios

Config/CorsConfig

Esta clase se encarga de gestionar con el Cors la entrada y el mapeo de rutas que podrán acceder a nuestra api teniendo en cuenta que no solo es nombrar las rutas sino que hemos de permitirlas en casos excepcionales además de la posibilidad de permitir métodos Http para más información visite la página de Cors .




```
@Configuration
class CorsConfig {
    @Bean
    fun corsConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            override fun addCorsMappings(registry: CorsRegistry) {
                registry.addMapping( pathPattern: "/**")
                    .allowedOrigins( ...origins: "http://localhost:8080")
                    .allowedHeaders( ...headers: "*")
                    .allowedMethods( ...methods: "*")
                    .maxAge(3600)
            }
```

Inyección de Dependencias

En este caso Spring cuenta con su propio sistema de inyección de dependencias a través de etiquetas y Beans algunos ejemplos son: @Configuration, @Service, @Configuration, @Autowired(aquí definimos las dependencias)

```
@Service
class ProductService
```

```
@Configuration
class CorsConfig {
```

```
@Autowired constructor(
    private var service: ProductService,
    private var tokenService: TokenService
) {
```

```
@Repository
interface ProductRepository: CoroutineCrudRepository<Product, Int>
```

Dtos/Mappers

Las clases dto´s en este programa están impuestas ya que referente al muestreo de datos al cliente no queremos que el cliente pueda visualizar ciertos datos al entrar y al muestreo de estos. Con lo cual referente a lo que devuelve cada ruta y entrada de datos a cada ruta utilizamos los dto´s.

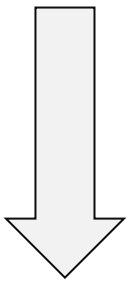
```
data class ServiceCreateDto(
    var category: String,
    var description: String,
    var price: Double,
    var url: String,
)
```

```
data class ProductDto(
    var name: String,
    var price: Double,
    var available: Boolean,
    var description: String,
    var url: String,
    var category: String,
    var stock: Int,
    var brand: String,
    var model: String
)
```

Mapeos Dto`s:

```
fun Product.toProductResponseDto(): ProductResponseDto {
    return ProductResponseDto(
        uuid = uuid,
        name = name,
        price = price,
        available = available,
        description = description,
        url = url,
        category = category.name,
        stock = stock,
        brand = brand,
        model = model
    )
}
```

```
fun Service.toServiceDto(): ServiceDto {
    return ServiceDto(
        category = this.category,
        description = this.description,
        price = this.price,
        url = this.url
    )
}
```



Models

En este caso hemos decidido ejecutar una herencia sobre Producto > Item < Servicio en este caso hemos decidido hacerlo con una clase abstracta, aunque en producción la clase Item no la guardamos dentro de la base de datos, sino que trabajamos con productos y servicios como clases junto con su atributo @Table que le da nombre a la clase en concepto de Base de Datos que están se guardarán dentro de la BBDD.

Repositories

En este caso hemos potado por una solución reactiva con lo cual hemos utilizado la librería R2BDC con la implementación de esta sobre MariaDB.
En Spring nos permite como interfaz pasarle un *CoroutineCrudRepository* para operar con reactividad tanto en la base de datos como en los repositorios del programa para mayor equilibrio de balanceo de datos.

```
@Repository
interface ProductRepository: CoroutineCrudRepository<Product, Int> {
    fun findProductsByCategory(category: String): Flow<Product>
    fun findProductByUuid(uuid: String): Flow<Product>
```

```
@Repository
interface ServiceRepository : CoroutineCrudRepository<Service, Int> {
    fun findServiceByUuid(uuid: String): Flow<Service>
}
```

Validators

Clases necesarias para comprobar la entrada de datos desde el exterior comprobando campos de los dto´s de entrada además de clases existen ciertos campos tageados las cuales nos permiten establecer valores por defecto, mínimos etc.

```
fun ServiceCreateDto.validate(): ServiceCreateDto {
    if (!ServiceCategory.values().map { it.toString() }.contains(this.category.uppercase()))
        throw ServiceBadRequestException("La categoría es incorrecta")
    if (this.description.isBlank())
        throw ServiceBadRequestException("La descripción no puede estar vacía")
    if (this.price <= 0)
        throw ServiceBadRequestException("El precio no puede ser igual o inferior a 0.")
    if (this.url.isBlank())
        throw ServiceBadRequestException("La url no puede estar vacía.")
    return this
```

```
@RequestBody
service: ServiceCreateDto,
ResponseEntity<Service> {
try {
    val roles = tokenService.
    if (roles.contains( other:
        service.validate()
```

Services

En el apartado de Services están la implementación de los métodos de los repositorios con la lógica de nuestro programa (implementado de esta manera ya que Spring cuenta con sus repositorios) y funciones como el update son modificadas con lo necesario para la lógica del programa.

```
@CachePut("products")
suspend fun updateProduct(product: Product, updateData: Product): Product {
    return repository.save(
        Product(
            id = product.id,
            uuid = product.uuid,
            name = updateData.name,
            price = updateData.price,
            available = updateData.available,
            description = updateData.description,
            url = updateData.url,
            category = updateData.category,
            stock = updateData.stock,
            brand = updateData.brand,
            model = updateData.model
        )
    )
}
```

```
suspend fun updateService(find: Services, service: ServiceUpdateDto): Services {
    return repository.save(
        Services(
            find.id,
            find.uuid,
            service.price,
            service.available,
            service.description,
            service.url,
            find.category
        )
    )
}
```

***TokenService**

Utilizamos dentro del programa un sistema de seguridad mediante tokens con base en JWT para la seguridad dentro de nuestro programa y comprobar que lo veremos más adelante: Generaremos el token mediante el login el cual se lo pasamos a las peticiones de entrada de la Api's ya que los métodos del programa están capados por el rol del usuario para mayor funcionalidad y hacer un sistema lo más parecido a la realidad. También utilizamos un algoritmo HMAC512(Código de autenticación mediante claves-hash) para la verificación del token.

```
@Service
class TokenService {
    @Value("\${jwt.secret}")
    private var secret: String? = null


    /**
     * Verificar el token.
     * @param token token a verifricar si es correcto.
     * @throws InvalidTokenException si el token es inválido.
     * @return el token verificado.
     */
    fun tokenVerify(token: String): DecodedJWT? {
        try {
            return JWT.require(Algorithm.HMAC512(secret)).build().verify(token)
        } catch (e: Exception) {
            throw InvalidTokenException("Token inválido o expirado")
        }
    }


    /**
     * Conseguir los claims de un token.
     * @param token token a saber sus claims.
     * @return los claims del token pedido.
     */
    fun getClaims(token: String) = tokenVerify(token)?.claims!!


    /**
     * Conseguir los roles de un token.
     * @param token token a saber sus roles.
     * @return los roles del token.
     */
    fun getRoles(token: String): String = getClaims(token)["roles"]!!.asString()

}
```

Caché

Para el apartado de la caché dentro del servicio de productos hemos implementado la cache propia del Framework de Spring que ya cuenta con métodos @Put, @Caheable, @CacheEvict. Vimos que ya que el propio Spring disponía de su sistema de cache no era necesario implementar una librería externa.

```
@CachePut("products")
suspend fun saveProduct(product: Product): Product {
```

```
@Cacheable("products")
suspend fun findProductById(id: Int): Product {
```

```
@CacheEvict("products")
suspend fun deleteProduct(uuid: String): Boolean {
```

Exceptions

¿Porque este apartado? En este apartado lo que hacemos es hacer excepciones personalizadas según la necesidad del problema ya que hacer una excepción tan generalizada no queda resolutiva y no es legible para el mantenimiento del código.

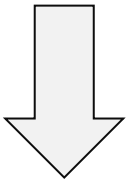
```
sealed class ProductException(message: String) : RuntimeException(message)
class ProductNotFoundException(message: String) : ProductException(message)
class ProductBadRequestException(message: String): ProductException(message)
```

```
    } catch (e: ProductNotFoundException) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    } catch (e: ProductBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    } catch (e: InvalidTokenException){
        throw ResponseStatusException(HttpStatus.UNAUTHORIZED, e.message)
    }
```

Storage

En este caso hemos implementado un servicio de almacenamiento para almacenar en este caso imágenes de productos mediante el sistema de Multiparte comprobado con tokens donde el mecanismo pensado es: **Para guardar:** que le entra una cadena multiparte dentro de los parámetros de entrada del api, luego de esto comprobamos que recibimos el token y dependiendo del rol podremos hacerlo. Luego de esto encontramos el producto al cual hará referencia la imagen y al hacer la lógica para salvar las imágenes junto con el producto. Y enviamos la respuesta.

```
@PostMapping(
    value = ["/product/{uuid}"],
    consumes = [MediaType.MULTIPART_FORM_DATA_VALUE]
)
fun storeProduct(
    @RequestHeader(HttpHeaders.AUTHORIZATION) token: String,
    @RequestPart("file") file: MultipartFile,
    @PathVariable uuid: String
): ResponseEntity<Map<String, String>> = runBlocking { this: CoroutineScope
    return@runBlocking try{
        val roles = tokenService.getRoles(token)
        if(roles.contains( other: "ADMIN") || roles.contains( other: "SUPERADMIN") || roles.contains( other: "EMPLEADO")) {
            productService.findProductByUuid(uuid)
            if (!file.isEmpty) {
                val saved = storageService.storeProduct(file, uuid)
                val response = mapOf("name" to saved, "created_at" to LocalDateTime.now().toString())
                ResponseEntity.status(HttpStatus.CREATED).body(response)
            } else {
                throw ResponseStatusException(HttpStatus.BAD_REQUEST, "No se puede almacenar un fichero vacío")
            }
        }else{
            throw ResponseStatusException(HttpStatus.UNAUTHORIZED, "No tienes permisos para realizar esta acción")
        }
    }catch (e: StorageBadRequestException){
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    }catch (e: ProductNotFoundException){
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    }catch (e: InvalidTokenException){
        throw ResponseStatusException(HttpStatus.UNAUTHORIZED, e.message)
    }
}
```



Para el Obtener: En este caso en el servicio de Storage tenemos un método para que mediante la entrada de un Path y que gracias a metidos incluidos en la clase Path y gracias a la implementación de la clase UriResource y comprobamos que existe la imagen y en caso de existir devolvemos el resource que dentro del método del controller el cual le comprueba y obtiene el resource convertido a absolutePath y devolvemos la respuesta con la imagen en caso de encontrarla.

```
@GetMapping(value = ["/product/{filename:.+}"])
@ResponseBody
fun getProductResource(
    @RequestHeader(HttpHeaders.AUTHORIZATION) token: String,
    @PathVariable filename: String?,
    request: HttpServletRequest
): ResponseEntity<Resource> = runBlocking { this: CoroutineScope
    val file: Resource = storageService.loadAsResource(filename.toString(), type: "PRODUCT")
    var contentType: String?
    contentType = try {
        request.servletContext.getMimeType(file.file.absolutePath)
    } catch (ex: IOException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, "No se puede determinar el tipo del fichero")
    }
    if (contentType == null) {
        contentType = "application/octet-stream"
    }
    return@runBlocking ResponseEntity.ok()
        .contentType(MediaType.parseMediaType(contentType))
        .body<Resource?>(file)
}
```

Delete:

```
@DeleteMapping(value = ["/product/{filename:.+}"])
@ResponseBody
fun deleteProductFile(@RequestHeader(HttpHeaders.AUTHORIZATION) token: String, @PathVariable filename: String?,
    request: HttpServletRequest
): ResponseEntity<Resource> = runBlocking { this: CoroutineScope
    return@runBlocking try {
        val roles = tokenService.getRoles(token)
        if(roles.contains( other: "ADMIN") || roles.contains( other: "SUPERADMIN") || roles.contains( other: "EMPLEADO")) {
            storageService.deleteProduct(filename.toString())
            ResponseEntity(HttpStatus.NO_CONTENT)
        }else{
            throw ResponseStatusException(HttpStatus.UNAUTHORIZED, "No tienes permisos para realizar esta acción")
        }
    } catch (e: StorageBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    } catch (e: InvalidTokenException){
        throw ResponseStatusException(HttpStatus.UNAUTHORIZED, e.message)
    }
}
```

Controllers

En este caso los controller son las distintas llamadas o rutas referentes a la api y la lógica que abarca a cada una. En spring utilizamos las etiquetas de @RestController: para indicarle que el controlador será orientado a ApiRest y @RequestMapping el cual guarda la ruta a la cual hace referencia las entradas a la api

```
@RestController
@RequestMapping("/api/product")
```

Un ejemplo de una entrada a la api desde esta clase cogeremos el @PostMapping (Hacemos un Post) y como en las otras consultas sobre nuestra api en spring que tenemos **como lógica que a cada consulta sobre la api desde nuestra Api General llegue un token el cual nos dice dentro del mismo el rol del usuario asociado al login dentro de la aplicación para filtrar acciones.**

```
@PostMapping("")
suspend fun saveProduct(
    @RequestHeader(HttpHeaders.AUTHORIZATION) token: String,
    @RequestBody dto: ProductDto
): ResponseEntity<ProductResponseDto> {
    return try {
        val roles = tokenService.getRoles(token)
        if(roles.contains( other: "ADMIN") || roles.contains( other: "SUPERADMIN") || roles.contains( other: "EMPLOYEE")){
            val product = dto.validate().toProduct()
            val created = service.saveProduct(product)
            ResponseEntity.status(HttpStatus.CREATED).body(created.toProductResponseDto())
        }else{
            throw ResponseStatusException(HttpStatus.UNAUTHORIZED,"No tienes permisos para realizar esta accion")
        }
    } catch (e: ProductBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    } catch (e: InvalidTokenException) {
        throw ResponseStatusException(HttpStatus.UNAUTHORIZED, e.message)
    }
}
```

Datos de interés

Al estar en un entorno de MariaDB tenemos un script sql el cual hace referencia a los modelos con la etiqueta @Table para el mapeo dentro de la base de datos.

```sql
CREATE TABLE IF NOT EXISTS products
(
    id          BIGINT AUTO_INCREMENT PRIMARY KEY,
    uuid        VARCHAR(255) UNIQUE,
    name        VARCHAR(255)    NOT NULL,
    price       DOUBLE PRECISION NOT NULL,
    available   BOOL            NOT NULL,
    description VARCHAR(255)    NOT NULL,
    url         VARCHAR(255)    NOT NULL,
    category    VARCHAR(255)    NOT NULL,
    stock       INTEGER         NOT NULL,
    brand       VARCHAR(255)    NOT NULL,
    model       VARCHAR(255)    NOT NULL
);
```

```sql
CREATE TABLE IF NOT EXISTS services
(
    id          BIGINT AUTO_INCREMENT PRIMARY KEY,
    uuid        VARCHAR(255) UNIQUE,
    price       DOUBLE PRECISION NOT NULL,
    available   BOOL            NOT NULL,
    description VARCHAR(255)    NOT NULL,
    url         VARCHAR(255)    NOT NULL,
    category    VARCHAR(255)    NOT NULL
);
```

Otro aspecto para recalcar es que utilizamos Docker para la aplicación para que desde cualquier sitio podamos ejecutar el programa con un contenedor de base de datos asociados, en este caso MariaDB.

```yaml
services:
  mariaDb:
    container_name: mariaDb
    image: mariadb
    environment:
      MARIADB_ROOT_PASSWORD: password
      MARIADB_USER: admin
      MARIADB_PASSWORD: 1234
      MARIADB_DATABASE : tienda
    ports:
      - "3306:3306"
    expose:
      - "3306:3306"
    volumes:
      - ./sql:/docker-entrypoint-initdb.d
```

```yaml
  api-productos:
    container_name: api-productos
    build: .
    ports:
      - "8082:8082"
    expose:
      - "8082:8082"
    depends_on:
      - mariaDb
    networks:
      - red
```

Usuarios

Config/Security

Esta api cuenta con un apartado muy importante que es el manejo de la seguridad el Spring ¿Por qué sobre esta api? Fácil. Spring Security cuenta con una extensión de clase la cual es UserDetails y UserDetailsService cada una diseñada para el manejo de la seguridad de entrada al realizar registers y logins con mayor facilidad y todo más controlado. Utiliza el JWT (Tratado de Tokens) el cual cuenta con varias clases las cuales tendremos que introducir que utilizará Security tales como el Authenticationfilter y el AuthorizationFilter.

Filtro de Autenticación:

```
class JwtAuthenticationFilter(
    private val jwtTokenUtil: JwtTokenUtil,
    private val authenticationManagerX: AuthenticationManager,
) : UsernamePasswordAuthenticationFilter() {

    /**
     * Attempt authentication
     * @param req
     * @param response
     * @return
     */
    override fun attemptAuthentication(req: HttpServletRequest, response: HttpServletResponse): Authentication {
        logger.info { "Intentando autenticar" }

        val credentials = ObjectMapper().readValue(req.inputStream, UsersLoginDto::class.java)
        val auth = UsernamePasswordAuthenticationToken(
            credentials.email,
            credentials.password,
        )
        return authenticationManagerX.authenticate(auth)
    }
```

```
override fun successfulAuthentication(
    req: HttpServletRequest?, res: HttpServletResponse, chain: FilterChain?,
    auth: Authentication
) {
    logger.info { "Autenticación correcta" }


    val user = auth.principal as Users
    val token: String = jwtTokenUtil.generateToken(user)
    res.addHeader( name: "Authorization", token)
    res.addHeader( name: "Access-Control-Expose-Headers", JwtTokenUtil.TOKEN_HEADER)
}
```

Filtro de Autorización:

```
class JwtAuthorizationFilter(
    private val jwtTokenUtil: JwtTokenUtil,
    private val service: UsersServices,
    authManager: AuthenticationManager,
) : BasicAuthenticationFilter(authManager) {
```

```
@Throws(IOException::class, ServletException::class)
override fun doFilterInternal(
    req: HttpServletRequest,
    res: HttpServletResponse,
    chain: FilterChain
) {
    val header = req.getHeader(AUTHORIZATION.toString())
    if (header == null || !header.startsWith(JwtTokenUtil.TOKEN_PREFIX)) {
        chain.doFilter(req, res)
        return
    }
    getAuthentication(header.substring( startIndex: 7))?.also { it: UsernamePasswordAuthenticationToken
        SecurityContextHolder.getContext().authentication = it
    }
    chain.doFilter(req, res)
}
```

```
private fun getAuthentication(token: String): UsernamePasswordAuthenticationToken? = runBlocking { this: CoroutineScope
    logger.info { "Obteniendo autenticación" }

    if (!jwtTokenUtil.isTokenValid(token)) return@runBlocking null
    // val username = jwtTokenUtil.getUsernameFromJwt(token)
    val userId = jwtTokenUtil.getUserIdFromJwt(token)
    // val roles = jwtTokenUtil.getRolesFromJwt(token)
    val user = service.loadUserbyUuid(userId)
    return@runBlocking UsernamePasswordAuthenticationToken(
        user,
        credentials: null,
        user?.authorities

    )
}
```

JwtTokenUtil:

Esta clase se encargará de con parámetros establecidos en el **application.properties** de Spring referenciados con las etiquetas necesarias de la creación del token junto con varios métodos de obtención de parámetros y validaciones.

```
#JWT Configuración de secreto y tiempo de token
jwt.secret=PracticaTiendaMusica?$
jwt.token-expiration=3600
```

```
@Component
class JwtTokenUtil {


    @Value("PracticaTiendaMusica?$")
    private val jwtSecreto: String? =
        null

    @Value("3600")
    private val jwtDuracionTokenEnSegundos = 0

    /**
     * Generate token
     *
     * @param user
     * @return
     */
    fun generateToken(user: Users): String {
        logger.info { "Generando token para el usuario: ${user.name}" }

        val tokenExpirationDate = Date( date: System.currentTimeMillis() + jwtDuracionTokenEnSegundos * 1000)

        return JWT.create()
            .withSubject(user.uuid)
            .withHeader(mapOf("typ" to TOKEN_TYPE))
            .withIssuedAt(Date())
            .withExpiresAt(tokenExpirationDate)
            .withClaim( name: "email", user.email)
            .withClaim( name: "name", user.name)
            .withClaim( name: "roles", user.rol.split( ...delimiters: ",").toSet().toString())

            .sign(Algorithm.HMAC512(jwtSecreto))
    }
```

```
fun validateToken(authToken: String): DecodedJWT? {
    logger.info { "Validando el token: ${authToken}" }

    try {
        return JWT.require(Algorithm.HMAC512(jwtSecreto)).build().verify(authToken)
    } catch (e: Exception) {
        throw TokenInvalidException("Token no válido o expirado")
    }
}
```

SecurityConfig

En esta clase está la lógica de la seguridad del programa. Lo que conlleva que tenga distintas etiquetas tales como:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true, prePostEnabled = true)
class SecutiryConfig
@Autowired constructor(
    private val userService: UsersServices,
    private val jwtTokenUtil: JwtTokenUtil
){
```

@EnableWebSecurity , @EnableMethodSecurity .No solo con usuarios sino con las entradas y lo que permite y no y trabajar con los endpoints dependiendo si queremos que utilice una ruta la cual esté abierta a todos con el atributo permitAll() o si no como spring security gracias al UserDetails lleva un sistema de roles los cuales se lo podemos especificar dentro de dicha clase. Y junto con esto a la cadena de filtro que es lo que se gestiona dentro de la clase le tendremos que añadir los filtros creados anteriormente junto con un authenticationManager. (Hay que recalcar que todo esto va de la mano con JWT). También recalcar que al introducir SpringSecurity las rutas se bloquearán a no ser que se introduzcan en el filterChain(Método build).

```
@Bean
fun filterChain(http:HttpSecurity):SecurityFilterChain{
    val authenticationManager=authManager(http)

    http
        .csrf() CsrfConfigurer<HttpSecurity!>!
        .disable() HttpSecurity!
        .exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity!>!
        .and() HttpSecurity!
        .authenticationManager(authenticationManager)
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSe
        .and() HttpSecurity!
        .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<HttpSecurity!>.AuthorizationManagerRequestMatcherRegistry!
        .requestMatchers( ...patterns: "/error/**").permitAll()
        .requestMatchers( ...patterns: "/api/users/login", "/api/users/register").permitAll()
        .requestMatchers( ...patterns: "/api/users/list").permitAll()//.hasAnyRole("EMPLOYEE","ADMIN","SUPERADMIN")
        .requestMatchers( ...patterns: "/api/users/me").hasAnyRole( ...roles: "USER","EMPLOYEE","ADMIN","SUPERADMIN")
        .and() HttpSecurity!
        .addFilter(JwtAuthenticationFilter(jwtTokenUtil, authenticationManager))
        .addFilter(JwtAuthorizationFilter(jwtTokenUtil, userService, authenticationManager))

    return http.build()
}
```

Controllers

En este caso los controller son las distintas llamadas o rutas referentes a la api y la lógica que abarca a cada una. En spring utilizamos las etiquetas de @RestController: para indicarle que el controlador será orientado a ApiRest y @RequestMapping el cual guarda la ruta a la cual hace referencia las entradas a la api.

```
@RestController
@RequestMapping(APIConfig.API_PATH + "/users")
```

Comprobamos que en este caso tenemos un login el cual cuenta con un atributo @Valid que también podemos especificar que este atributo hace referencia a la clase Validators la cuan tiene como objetivo que la información entre bien y el método login devuelve un usuario con token que es el token que utilizaremos como base para realizar las llamadas a las demás Api´s.

```
@PostMapping("/login")
fun login(@Valid @RequestBody logingDto: UsersLoginDto): ResponseEntity<UsersWithTokenDto> {
    logger.info { "Login:${logingDto.email}" }
    val authentication: Authentication = authenticationManager.authenticate(
        UsernamePasswordAuthenticationToken(
            logingDto.email,
            logingDto.password
        )
    )
    SecurityContextHolder.getContext().authentication = authentication
    val user = authentication.principal as Users
    val jwtToken: String = jwtTokenUtil.generateToken(user)
    logger.info { "Token: ${jwtToken}" }
    val userWithToken = UsersWithTokenDto(user.toDto(), jwtToken)
    return ResponseEntity.ok(userWithToken)
}
```

Varios métodos de los usuarios hacen referencia a según el rol que conllevan con lo cual existe una etiqueta la cual se llama **@PreAuthorize** la cual le pasamos la instrucción de los roles que queremos que accedan a este método.

```
@PreAuthorize("hasAnyRole('USER','EMPLOYEE','ADMIN','SUPERADMIN')")
@GetMapping("/me")
fun meInfo(@AuthenticationPrincipal user: Users): ResponseEntity<UsersDto> {

    logger.info { "Obteniendo la informacíon  del usuario : ${user.name}" }

    return ResponseEntity.ok(user.toDto())
}
```

```
@PreAuthorize("hasAnyRole('EMPLOYEE','ADMIN','SUPERADMIN')")
@GetMapping("/list")
suspend fun list(@AuthenticationPrincipal user: Users): ResponseEntity<List<UsersDto>> {

    logger.info { "Obteniendo lista de usuarios" }

    val res = usersService.findAll().toList().map { it.toDto() }
    return ResponseEntity.ok(res)
}
```

Otro apunte es que implementamos el servicio de storage antes explicado aquí también, pero para el manejo de los avatares.

```
@PatchMapping(value = ["/me"], consumes = [MediaType.MULTIPART_FORM_DATA_VALUE]
) suspend fun updateAvatar(@AuthenticationPrincipal user: Users, @RequestPart("file") file: MultipartFile): ResponseEntity<UsersDto> {
    logger.info { "Actualizando avatar ${user.username}" }
    try {
        var urlImagen = user.url
        if (!file.isEmpty) {
            val imagen: String = storageService.save(file, user.uuid)
            urlImagen = storageService.getUrl(imagen)
        }
        val userAvatar = user.copy(
            url = urlImagen
        )
        val userUpdated = usersService.update(userAvatar)
        return ResponseEntity.ok(userUpdated.toDto())
    } catch (e: UsersBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    } catch (e: StorageException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    }
}
```

Dtos/Mappers

Las clases dto´s en este programa están impuestas ya que referente al muestreo de datos al cliente no queremos que el cliente pueda visualizar ciertos datos al entrar y al muestreo de estos. Con lo cual referente a lo que devuelve cada ruta y entrada de datos a cada ruta utilizamos los dto´s.

```kotlin
data class UsersDto(
    val uuid:String,
    val email: String,
    val name: String,
    val telephone: String,
    val url: String,
    val rol: Set<String> = setOf(Users.TypeRol.USER.name),
    val metadata: MetaData? = null,
) {
    data class MetaData(
        val createdAt: LocalDateTime? = LocalDateTime.now(),
        val updatedAt: LocalDateTime? = LocalDateTime.now(),
        val deleted: Boolean = false
    )
```

```kotlin
data class UsersWithTokenDto(
    val user: UsersDto,
    val token: String
)
/**
 * dto para el login de usuario.
 */
data class UsersLoginDto(
    val email: String,
    val password: String
)
```

Exceptions

¿Porque este apartado? En este apartado lo que hacemos es hacer excepciones personalizadas según la necesidad del problema ya que hacer una excepción tan generalizada no queda resolutiva y no es legible para el mantenimiento del código. En este caso como apunte podemos directamente sobre la excepción creada establecer un HttpStatus con la etiqueta **@ResponseStatus** Junto con el tipo que deseamos.

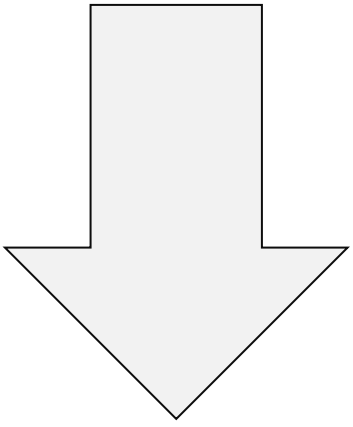
```kotlin
/**
 * Excepción específica, código 404 no encontrado.
 */
@ResponseStatus(HttpStatus.NOT_FOUND)
class UsersNotFoundException(message: String) : RuntimeException(message)

/**
 * Excepción específica, código 400, error percibido del cliente.
 */
@ResponseStatus(HttpStatus.BAD_REQUEST)
class UsersBadRequestException(message: String) : RuntimeException(message)
```

Models

Este apartado es muy importante ya que de esto depende Spring Security ya que contamos con la clase **User** que a su vez extiende de **UserDetails** la clase necesaria para el manejo de usuarios, la cual nos generará ciertos métodos necesarios . ¿Qué conlleva esto? que dentro de la misma hay ciertos parámetros los cuales han de estar sí o sí. Tales como: **el rol, el password y el name.**

```
@Table(name = "users")
data class Users(
    @Id
    val id :Int?=null,
    @Column("uuid")
    val uuid:String= UUID.randomUUID().toString(),
    @Column("email")
    val email:String,
    @Column("name")
    val name:String,
    @get:JvmName( name: "userPassword")
    @Column("password")
    val password:String,
    @Column("telephone")
    val telephone:Int = 787744741,
    @Column("rol")
    val rol :String=TypeRol.USER.name,
    @Column("avaliable")
    val avaliable:Boolean=true,
    @Column("url")
    val url:String="",
    @Column("created_at")
    val createdAt: LocalDateTime = LocalDateTime.now(),
    @Column("updated_at")
    val updatedAt: LocalDateTime = LocalDateTime.now(),
    val deleted: Boolean = false,
): UserDetails {
```

```
enum class TypeRol() {
    USER,EMPLOYEE,ADMIN,SUPERADMIN
}

override fun getAuthorities():
        MutableCollection<out GrantedAuthority> {

    return rol.split( ...delimiters: ",")
        .map { SimpleGrantedAuthority( role: "ROLE_${it.trim()}") }.toMutableList()
}

override fun getPassword(): String {
    return password
}

override fun getUsername(): String {
    return name
}

override fun isAccountNonExpired(): Boolean {
    return true
}

override fun isAccountNonLocked(): Boolean {
    return true
}

override fun isCredentialsNonExpired(): Boolean {
    return true
}

override fun isEnabled(): Boolean {
    return true
}
```



Repositories

En este caso hemos potado por una solución reactiva con lo cual hemos utilizado la librería R2BDC con la implementación de esta sobre MariaDB.
En Spring nos permite como interfaz pasarle un *CoroutineCrudRepository* para operar con reactividad tanto en la base de datos como en los repositorios del programa para mayor equilibrio de balanceo de datos.

```
@Repository
interface UsersRepository : CoroutineCrudRepository<Users, Long> {
```

Servicio de Uusarios:

```
@Service
class UsersServices
@Autowired constructor(
    private val repository: UsersRepository,
    private val passwordEncoder: PasswordEncoder
): UserDetailsService {
```

Clase que encripta la contraseña gracias a BCRYPT

```
suspend fun save(user: Users, isAdmin: Boolean = false): Users = withContext(Dispatchers.IO) { this: CoroutineScope
    logger.info { "Guardando usuario: $user" }
    if (repository.findByEmail(user.email).firstOrNull() != null) {
        throw UsersBadRequestException("El usuario ya existe con este email")
    }
    if (repository.findByTelephone(user.telephone).firstOrNull() != null) {
        throw UsersBadRequestException("El usuario ya existe con este numero de telefono ")
    }
    var newUser = user.copy(
        uuid = UUID.randomUUID().toString(),
        password = passwordEncoder.encode(user.password),
        rol = Users.TypeRol.USER.name,
        createdAt = LocalDateTime.now(),
        updatedAt = LocalDateTime.now()
    )
    if (isAdmin) {
        newUser = newUser.copy(
            rol = Users.TypeRol.ADMIN.name
        )
    }
    println(newUser)
    try {
        return@withContext repository.save(newUser)
    } catch (e: Exception) {
        throw UsersBadRequestException("Error al crear el usuario: Nombre de usuario o email ya existen")
    }
```

Validators

Clases necesarias para comprobar la entrada de datos desde el exterior comprobando campos de los dto´s de entrada además de clases existen ciertos campos tageados las cuales nos permiten establecer valores por defecto, mínimos etc.

```
/**
 * Método que valida los datos para la creación de los usuarios.
 */
fun UsersCreateDto.validate(): UsersCreateDto {
    if (this.name.isBlank()) {
        throw UsersBadRequestException("El nombre no puede estar vacío")
    } else if (this.email.isBlank() || !this.email.matches(Regex( pattern: "^[A-Za-z0-9+_.-]+@(.+)\$")))
        throw UsersBadRequestException("El email no puede estar vacío o no tiene el formato correcto")
    else if (this.password.isBlank() || this.password.length < 5)
        throw UsersBadRequestException("El password no puede estar vacío o ser menor de 5 caracteres")
    return this
}

/**
 * Método que valida los datos para la actualización de usuarios.
 */
fun UsersUpdateDto.validate(): UsersUpdateDto {
    if (this.email.isBlank() || !this.email.matches(Regex( pattern: "^[A-Za-z0-9+_.-]+@(.+)\$")))
        throw UsersBadRequestException("El email no puede estar vacío o no tiene el formato correcto")
    else if (this.name.isBlank())
        throw UsersBadRequestException("El name no puede estar vacío")
    else if (this.telephone.isBlank())
        throw UsersBadRequestException("El telephone no puede estar vacio o ser menor de 5 caracteres")
    return this
}
```

SSL

En esta api hemos utilizado un sistema ssl para mayor seguridad ya que lo vimos como parametro obligatorio tratandose de datos personales de usuarios.

```
# SSL Problema con la validación del certificado
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:cert/tiendaMusica_keystore.p12
server.ssl.key-store-password=1234567
server.ssl.key-alias=tiendaMusicaKeyPair
server.ssl.enabled=false
```

Fragmento sacado del application.properties

Pedidos

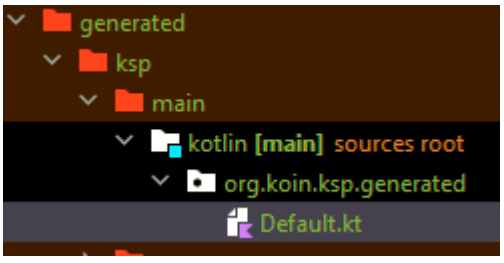
Aquí ya cambiamos de Framework pasamos a la arquitectura de Ktor + Mongo. ¿Por qué? Como ya explicamos en la arquitectura del sistema consideramos que para el manejo de pedidos una estructura NoSql podría ser beneficiosa porque nos permite manejar un gran volumen de datos y nos facilita la tarea de por ejemplo el embeber o referenciar ciertos campos. En esta Api de Pedidos optamos que un parámetro descriptivo como es el usuario lo referenciamos mediante su id.

Controllers

Clase que gestiona los métodos que le brindamos desde el repositorio caprturando excepciones en caso de haberlas claro.


```
@Single
class OrderController(private val repository: OrderRepository) {

    suspend fun createOrder(order: Order): Order {
        logger.debug( msg: "Trying to find User...")
        try {
            logger.debug( msg: "Creating Order: ${order.uuid}")
            return repository.save(order)
        } catch (e: Exception) {
            throw Exception(e.message!!)
        }
    }

    suspend fun patchOrder(order: Order, newData: OrderUpdateDto): Order {
        logger.debug { "Patching pedido" }
        if (newData.price != null)
            order.price = newData.price
        if (newData.status != null)
            order.status = newData.status

        return repository.save(order)
    }
```

Inyección de Dependencias



Como esto no es Spring en este caso lo que hemos optado es a utilizar una librera utilizada anteriormente en nuestro proyectos la cual es Koin pero en este caso en vez de contar con que nosotros a mano hagamos la inyección dejamos que mediante etiquetas tales como **@Single** en el apartado de plugins que se divisará mas adelante estará su configuración y el archivo lo creará Koin en sí. (**Al hacer el build**).

Dto´s/Mappers

Las clases dto´s en este programa están impuestas ya que referente al muestreo de datos al cliente no queremos que el cliente pueda visualizar ciertos datos al entrar y al muestreo de estos. Con lo cual referente a lo que devuelve cada ruta y entrada de datos a cada ruta utilizamos los dto´s.

```
@Serializable
data class OrderDto(
    val id: String,
    @Serializable(UUIDSerializer::class)
    val uuid: UUID,
    val price: Double,
    val user: UserDto,
    val status: Order.Status,
    @Serializable(LocalDateSerializer::class)
    val createdAt: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val deliveredAt: LocalDate?
)
```

```
@Serializable
data class OrderCreateDto(
    val price: Double,
    val products: List<SellLine>,
    val userId: String
)
```

Exceptions

¿Porque este apartado? En este apartado lo que hacemos es hacer excepciones personalizadas según la necesidad del problema ya que hacer una excepción tan generalizada no queda resolutiva y no es legible para el mantenimiento del código. En este caso lamentablemente ktor no cuenta con un sistema de etiquetas para indicarle el status.

```
sealed class OrderException(message: String) : RuntimeException(message)


class OrderNotFoundException(message: String) : OrderException(message)
class OrderBadRequest(message: String) : OrderException(message)
class OrderUnauthorized(message: String) : OrderException(message)
class OrderDuplicated(message: String) : OrderException(message)
```

Models

Los modelos en este caso hemos de tener en cuenta que irá referenciado a una base de datos en Mongo con lo cual etiquetas tales como **@BsonId y @Contextual** son necesarias.

```
@Serializable
data class Order(
    @BsonId @Contextual
    val id: String = newId<Order>().toString(),
    val uuid: String = UUID.randomUUID().toString(),
    var price: Double,
    var userId: String,
    var status: Status,
    @Serializable(LocalDateSerializer::class)
    var createdAt: LocalDate,
    @Serializable(LocalDateSerializer::class)
    var deliveredAt: LocalDate?,
    val productos: MutableList<SellLine>
) {
```

Repositories

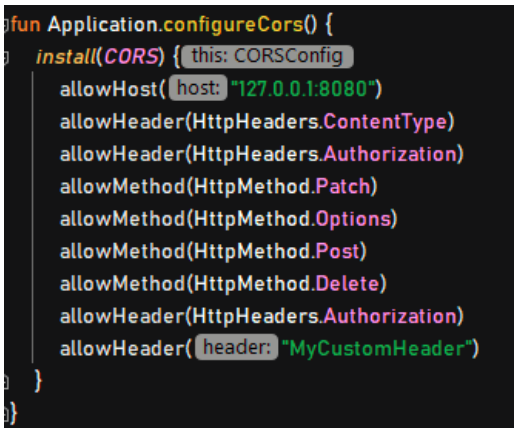
En este caso hemos de crear nosotros los métodos del repositorio a través del MongoDbManager con los métodos necesarios en nuestro programa en este caso un CRUD.

```
@Single
class OrderRepository : IOrderRepository {
    override suspend fun findAll(): Flow<Order> {
        return MongoDbManager.mongoDatabase.getCollection<Order>()
            .find().publisher.asFlow()
    }

    fun findAll(page: Int, perPage: Int): Flow<Order> {
        return MongoDbManager.mongoDatabase.getCollection<Order>() CoroutineCollection<Order>
            .find() CoroutineFindPublisher<Order>
            .skip(page * perPage)
            .limit(perPage)
            .publisher.asFlow()
    }

    override suspend fun findById(id: Id<Order>): Order? {
        return MongoDbManager.mongoDatabase.getCollection<Order>().findOneById(id)
    }


    override suspend fun save(entity: Order): Order {
        return MongoDbManager.mongoDatabase.getCollection<Order>().save(entity).let { entity }
    }


    override suspend fun delete(entity: Order): Boolean {
        return MongoDbManager.mongoDatabase.getCollection<Order>().deleteOneById(entity.id).let { true }
    }

    override suspend fun findByUser(id: String): Flow<Order> {
        return MongoDbManager.mongoDatabase.getCollection<Order>().find( filter: "{userId: '$id'}").publisher.asFlow()
    }
```

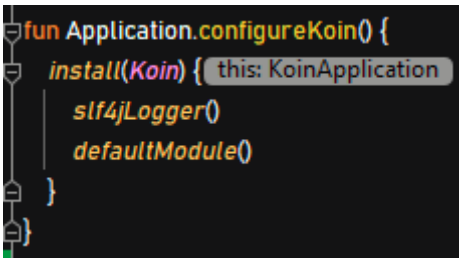
Plugins

La parte más importante ya que ktor trabaja con la implementación y configuración de plugins para su funcionamiento que son todas las utilidades que nuestro programa va a utilizar, configuramos los plugins mediante funciones de extensión sobre **Application** . Las que hemos utilizado son las siguientes: **Cors, Koin, Routing, Serialization, Validators.**

Cors

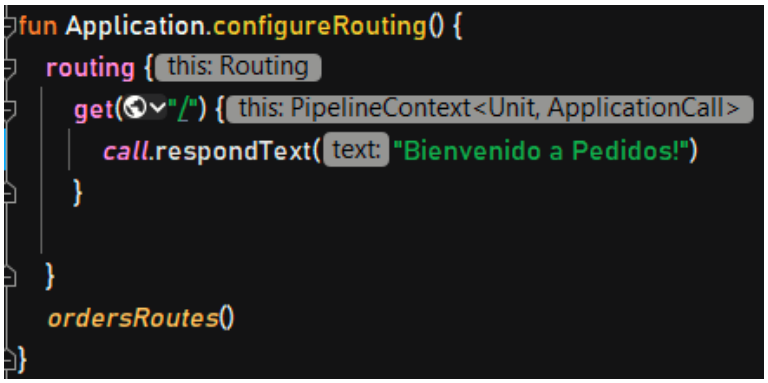


Koin



Routing

En este apartado gestionamos la ruta origen en nuestro caso es / ya que de ahí saldrán las rutas y junto a ella está un método que es **orderRoutes()** que es necesario para mapear las rutas de pedidos si no se encuentra dentro de este método las rutas no funcionan.



Serialization

```
fun Application.configureSerialization() {
    install(ContentNegotiation) { this: ContentNegotiationConfig
        json()
    }
    routing { this: Routing
        get("/json/kotlinx-serialization") { this: PipelineContext<Unit, ApplicationCall>
            call.respond(mapOf("Esto es" to "Serialization"))
        }
    }
}
```

Validators

También tendremos que indicarle la clase de validación dentro del plugin para que la configure junto con la aplicación.

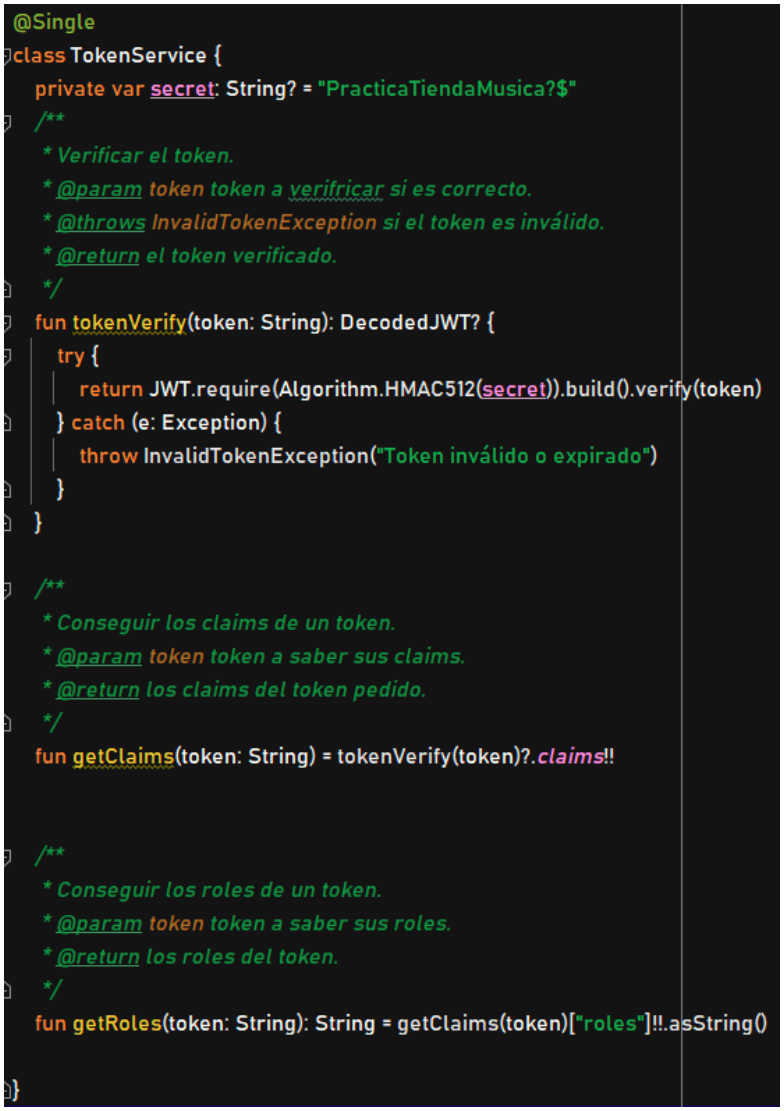
```
fun Application.configuteValidations(){
    install(RequestValidation){ this: RequestValidationConfig
        orderValidators()
    }

}
```

Services

MongoDbManager es la clase que configura todo los parámetros del mongo sobre la cual por cierto mongo trabajamos con mongo con corrutinas.

```
object MongoDbManager {
    private var mongoClient: CoroutineClient
    var mongoDatabase: CoroutineDatabase

    init {
        val property= Property( nameFile: "mongo.properties")
        println(property.getKey( key: "mongoDb.uri"))
        mongoClient = KMongo.createClient(property.getKey( key: "mongoDb.uri"))
            .coroutine
        println("Conectado")
        mongoDatabase = mongoClient.getDatabase(property.getKey( key: "mongoDb.databaseName"))
    }
}
```

TokenService clase la cual utiliza el JWT para en este caso verificar un token entrante que como hemos planteado desde la general nos entrará un rol a través de un token lo que conlleva a que sea verificado y que los claims del token se verifiquen donde dentro de los mismos.



```
@Single
class TokenService {
    private var secret: String? = "PracticaTiendaMusica?$"
    /**
     * Verificar el token.
     * @param token token a verifricar si es correcto.
     * @throws InvalidTokenException si el token es inválido.
     * @return el token verificado.
     */
    fun tokenVerify(token: String): DecodedJWT? {
        try {
            return JWT.require(Algorithm.HMAC512(secret)).build().verify(token)
        } catch (e: Exception) {
            throw InvalidTokenException("Token inválido o expirado")
        }
    }

    /**
     * Conseguir los claims de un token.
     * @param token token a saber sus claims.
     * @return los claims del token pedido.
     */
    fun getClaims(token: String) = tokenVerify(token)?.claims!!


    /**
     * Conseguir los roles de un token.
     * @param token token a saber sus roles.
     * @return los roles del token.
     */
    fun getRoles(token: String): String = getClaims(token)["roles"]!!.asString()

}
```

Routes

En este caso como es un apartado de rutas hacemos una función de extensión sobre aplicación. Sobre la misma inyectamos el controlador y el servicio de tokens para comprobar los roles .

Lógica de la ruta: Entra la opción junto con su token, una vez entre el token se verifica si ha llegado y entonces una vez tengamos el token mediante el TokenService obtenemos los roles y dependiendo del rol opera sobre la ruta y hace la opción.

```
routing { this: Routing
  route(ENDPOINT) { this: Route

    //Recupera todos los pedidos guardados.
    get { this: PipelineContext<Unit, ApplicationCall>
      val token = call.request.headers["Authorization"]?.replace( oldValue: "Bearer ", newValue: "").toString()
      token?.let { it: String
        logger.debug { "GET ALL $ENDPOINT" }
        val roles = tokensService.getRoles(it)
        if (roles.contains( other: "ADMIN") || roles.contains( other: "SUPERADMIN") || roles.contains( other: "EMPLOYEE")) {
          val page = call.request.queryParameters["page"]?.toIntOrNull()
          val perPage = call.request.queryParameters["perPage"]?.toIntOrNull() ?: 10
          if (page != null && page > 0) {
            val res = ordersService.getAllOrders( page: page - 1, perPage) Flow<Order>
              .toList() List<Order>
              .map { it.toDto() } List<OrderDto>
              .let { res -> call.respond(HttpStatusCode.OK, OrderPageDto(page, perPage, res)) }
          }
          ordersService.getAllOrders().toList().let { res -> call.respond(HttpStatusCode.OK, res) }
        }
      }
    }
```

```
patch("{id}") { this: PipelineContext<Unit, ApplicationCall>
  logger.debug { "PATCH ORDER : $ENDPOINT/{id}" }
  val token = call.request.headers["Authorization"]?.replace( oldValue: "Bearer ", newValue: "").toString()
  token?.let { it: String
    val roles = tokensService.getRoles(token)
    if (roles.contains( other: "SUPERADMIN")) {
      try {
        val dto = call.receive<OrderUpdateDto>()
        val id = call.parameters["id"]
        val pedido = ordersService.getById(id!!)
        pedido?.let { it: Order
          ordersService.patchOrder(pedido, dto)
          call.respond(HttpStatusCode.OK)
        } ?: run { this: PipelineContext<Unit, ApplicationCall>
          call.respond(HttpStatusCode.NotFound, message: "Not found")
        }
      } catch (e: OrderUnauthorized) {
        call.respond(HttpStatusCode.Unauthorized, e.message.toString())
      } catch (e: OrderDuplicated) {
        call.respond(HttpStatusCode.Conflict, e.message.toString())
      }
    }
  }
```

Validation

Clase de validación la cual comprueba los datos de entrada para que entren de forma correcta dentro del programa,

```
fun RequestValidationConfig.orderValidators(){
    validate<OrderCreateDto>{ dto ->
    if (dto.price <=0){
        throw OrderBadRequest(" El precio del pedido no puede ser menor o igual a 0")
    }else if (dto.userId.isBlank()){
        throw  OrderBadRequest(" El id del usuario  no puede estar en blanco")
    }else if(dto.products.isEmpty()){
        throw OrderBadRequest (" La lista de productos no puede estar vacía")
    }
    else{
        ValidationResult.Valid
    }
    }
    validate<OrderUpdateDto>{ dto ->
        if(dto.price!! <= 0){
            throw OrderBadRequest("El precio del pedido no puese der menor o igual a 0")

        }else {
            ValidationResult.Valid
        }
    }
}
```

Application.kt

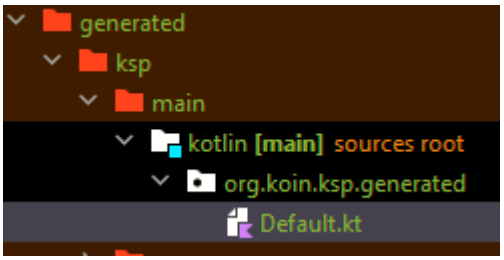
Este apartado es muy importante ya que dentro de sí debemos tener una función de extensión sobre module donde dentro de sí hemos de llamar a los plugins que hemos definido en la configuración.

```
fun main(args: Array<String>): Unit {
    println("Hola")
    EngineMain.main(args)
}

@Suppress( ...names: "unused")
fun Application.module() = runBlocking { this: CoroutineScope
    configureKoin()
    configureSerialization()
    configureRouting()
    configureCors()
    configuteValidations()
    sampleData()
}
```

ApiGeneral

La api más importante de todas la cual hace de intermediaria entre las demas ya que gracias a esta maneja el envío del token y gestiona la entrada a las llamadas de las Api´s de forma segura al contar con SSL donde la lógica es la siguiente: **Desde la Api de usuarios al loguearnos le pasamos el token a cada ruta de la general donde a cada llamada le pasaremos el token y mediante Retrofit pasandole el token mediante el Header a las rutas.**

Inyección de Dependencias



Como esto no es Spring en este caso lo que hemos optado es a utilizar una librera utilizada anteriormente en nuestro proyectos la cual es Koin pero en este caso en vez de contar con que nosotros a mano hagamos la inyección dejamos que mediante etiquetas tales como **@Single** en el apartado de plugins que se divisará más adelante estará su configuración y el archivo lo creará Koin en sí. (**Al hacer el build**).

Dto´s

Las clases dto´s en este programa están impuestas ya que cada clase que se refleja sobre la Api como cada dato entrante es un Dto´s tendremos modelos dentro de la general pero solo serán los dto´s de entrada. Ya que al hacer una referencia entre Api´s le pasamos los Dto´s.

```
@Serializable
data class OrderCreateDto(
    val price: Double,
    val products: List<SellLine> = mutableListOf(),
    val userId: String
)
```

```
@Serializable
data class UserLoginDto(
    val email:String,
    val password:String
)
```

Exceptions

¿Porque este apartado? En este apartado lo que hacemos es hacer excepciones personalizadas según la necesidad del problema ya que hacer una excepción tan generalizada no queda resolutiva y no es legible para el mantenimiento del código. En este caso lamentablemente ktor no cuenta con un sistema de etiquetas para indicarle el statu.

```
sealed class OrderException(message: String) : RuntimeException(message)


class OrderNotFoundException(message: String) : OrderException(message)
class OrderBadRequest(message: String) : OrderException(message)
class OrderUnauthorized(message: String) : OrderException(message)
class OrderDuplicated(message: String) : OrderException(message)
```

Plugins

La parte más importante ya que ktor trabaja con la implementación y configuración de plugins para su funcionamiento que son todas las utilidades que nuestro programa va a utilizar, configuramos los plugins mediante funciones de extensión sobre **Application** . Las que hemos utilizado son las siguientes: **Cors(Http), Koin, Routing, Serialization, Validators.**

Cors(Http)

En este caso como es la general permitimos los host pero tendremos que recalcarle al Cors que permita ciertos métodos ya que será necesario sino no podremos desde la general llamar a los métodos **Get, Post, Patch etc.**

```
fun Application.configureHTTP() {
    install(CORS) { this: CORSConfig
        allowMethod(HttpMethod.Options)
        allowMethod(HttpMethod.Get)
        allowMethod(HttpMethod.Post)
        allowMethod(HttpMethod.Put)
        allowMethod(HttpMethod.Delete)
        allowMethod(HttpMethod.Patch)
        allowHeader(HttpHeaders.Authorization)
        allowHeader( header: "MyCustomHeader")
        anyHost()
    }
```

Koin

En este apartado creamos los modelos porque fallaba en el inyectar y lo que hacemos simplemente le pasamos la clase named para referenciar esos módulos y también con el **defaultModule()** para las otras etiquetas del programa.

```
fun Application.configureKoin() {
    install(Koin) { this: KoinApplication
        slf4jLogger()
        defaultModule()
        modules(moduleApp)
        modules(moduleApp2)
        modules(moduleApp3)
    }
}

val moduleApp = module { this: Module
    single<RetroFitRest>(named( name: "apiProduct")) { RetroFitClient().getInstance(RetroFitClient.API_PRODUCT) }
}
val moduleApp2 = module { this: Module
    single<RetroFitRestPedidos>(named( name: "apiOrder")) { RetroFitClientOrder().getInstance(RetroFitClientOrder.API_ORDER) }
}
val moduleApp3 = module { this: Module
    single<RetroFitRest>(named( name: "apiUsuarios")) { RetroFitClient().getInstance(RetroFitClient.API_USERS) }
```

Routing

En este apartado gestionamos la ruta origen en nuestro caso es / ya que de ahí saldrán las rutas y junto a ella está un método que son las **rutas** que es necesario para mapear las rutas de pedidos si no se encuentra dentro de este método las rutas no funcionan.

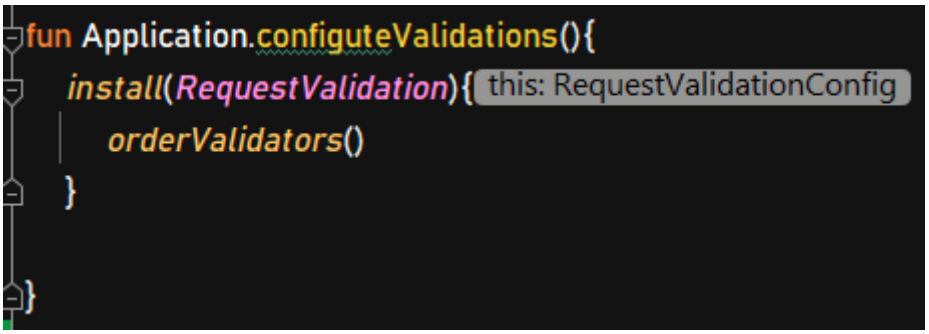
```
fun Application.configureRouting() {
    routing { this: Routing
        get("/") { this: PipelineContext<Unit, ApplicationCall>
            call.respondText( text: "Bienvenido a la Api General!")
        }
    }
    serviciosRoutes()
    productsRoutes()
    orderRoutes()
    usuariosRoutes()
}
```

Serialization

```
fun Application.configureSerialization() {
    install(ContentNegotiation) { this: ContentNegotiationConfig
        json(Json { this: JsonBuilder
            prettyPrint = true
            isLenient = true
            ignoreUnknownKeys = true
            coerceInputValues = true
        })
    }
}
```

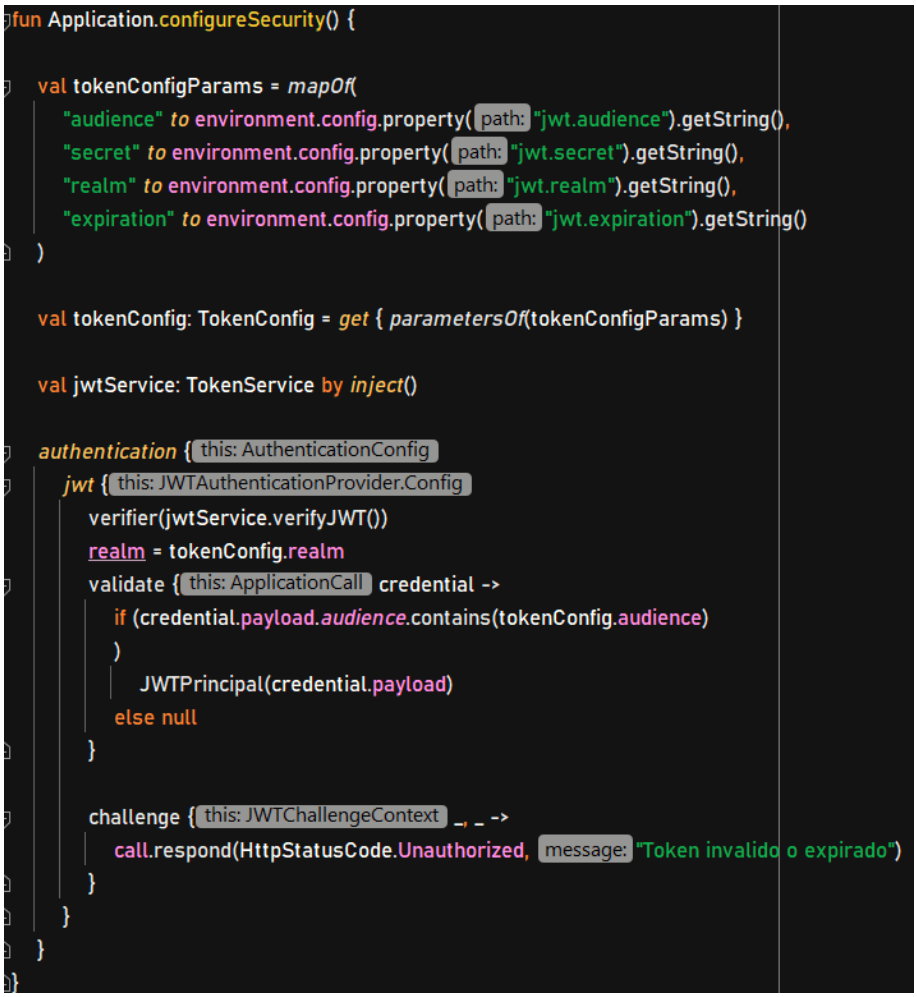
Validators

También tendremos que indicarle la clase de validación dentro del plugin para que la configure junto con la aplicación.



Security

En este plugin lo que hacemos es que comprobamos el token que nos entra para que se autentique el token antes de enviarle el token a las demás Api´s



Services

RetrofitRest en este apartado hemos decidido Retrofit ya que tenemos mas generalizado e interiorizado para el llamado de las rutas dependiendo de la ruta raíz hacemos distintas cosas, también hemos de crear el cliente de Retrofit con las rutas de las Api´s a las que la general está accediendo.

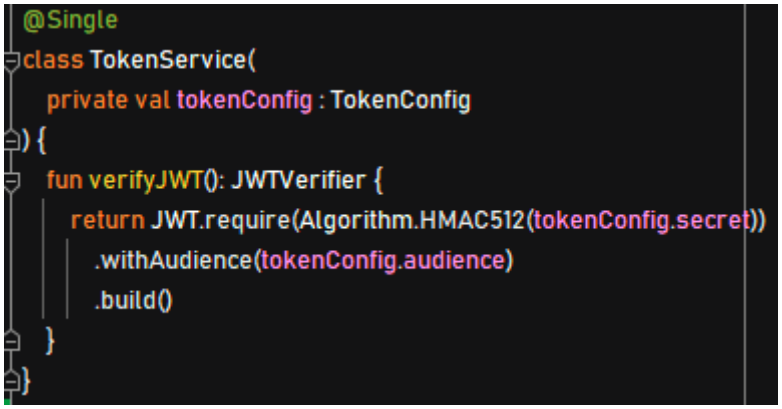
```
class RetroFitClient {

    companion object{
        const val API_PRODUCT = "http://api-productos:8082"
        const val API_USERS = "http://localhost:8083"
    }

    @Single
    @OptIn(ExperimentalSerializationApi::class)
    fun getInstance(url: String): RetroFitRest {
        val contentType = MediaType.get("application/json")
        return Retrofit.Builder().baseUrl(url) Retrofit.Builder
            .addConverterFactory(Json.asConverterFactory(contentType))
            .build() Retrofit
            .create(RetroFitRest::class.java)
    }
```

Llamadas de Ejemplo:

```
@POST("/api/users/register")
suspend fun registerUser(@Body user : UserCreateDto) : Response<UserTokenDto>
@POST("/api/users/login")
suspend fun loginUser(@Body user : UserLoginDto) : Response<UserTokenDto>


@GET("/api/users/list")
suspend fun getAllUsers(@Header("Authorization") token: String) : Response<List<UserDto>>

@GET("/api/users/me")
suspend fun getUserMe(@Header("Authorization") token : String) : Response<UserDto>
/**
```

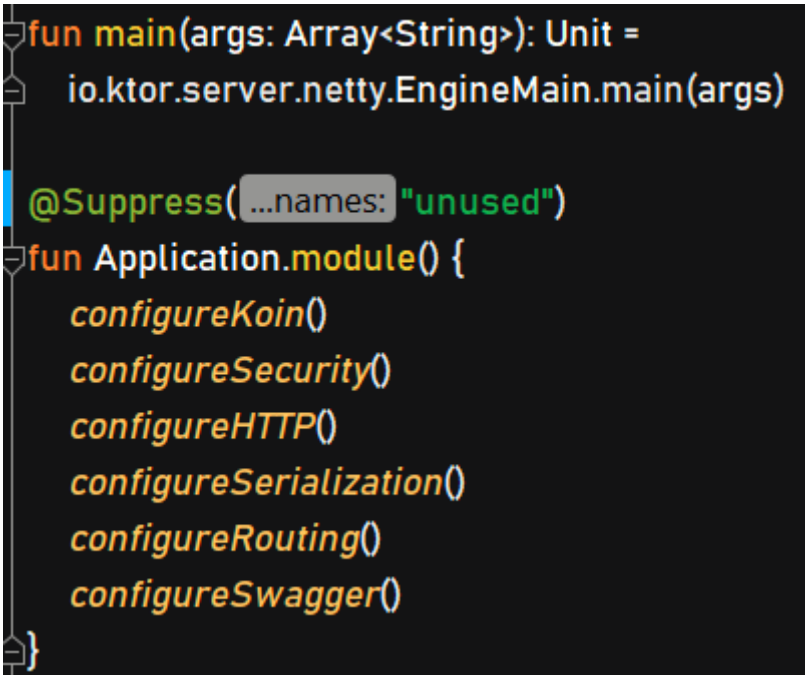
```
@Multipart
@POST("/api/storage/product/{id}")
suspend fun saveFileProduct(@Path("id") id: String, @Header("Authorization") token: String, @Part file: MultipartBody.Part): Response<Map<String, String>>

@GET("/api/storage/product/{filename}")
suspend fun getFileProduct(@Path("filename") filename: String, @Header("Authorization") token: String): ResponseBody

@DELETE("/api/storage/product/{filename}")
suspend fun deleteFileProduct(@Path("filename") filename: String, @Header("Authorization") token: String): Response<Void>
```

TokenService clase la cual utiliza el JWT para en este caso verificar un token entrante que como hemos planteado desde la general nos entrará un rol a través de un token lo que conlleva a que sea verificado y que los claims del token se verifiquen donde dentro de los mismos.



Application.kt

Este apartado es muy importante ya que dentro de sí debemos tener una función de extensión sobre module donde dentro de sí hemos de llamar a los plugins que hemos definido en la configuración.



Routes

En este caso como es un apartado de rutas hacemos una función de extensión sobre aplicación. Pero sobre esta lo que haremos ahora inyectarle el cliente Retrofit el cual tendrá las llamadas de las demás Api´s.
APUNTE IMPORTANTE: Las llamadas a otra api están en un sistema de respuesta con lo cual hemos de lanzar la consulta en una corrutina ya que de no hacerlo el tiempo de respuesta supera al establecido y la consulta no llega a su destino. Por eso veremos que en todos los métodos cuando hacemos la llamada del Retrofit le introducimos un async/await.
Lógica de la ruta: En este caso como utilizamos el token en todas para comprobar tema de roles etc. Simplemente comprobamos que en los parámetros de entrada de la general en los headers se encuentre el token. En casos puntuales como los posts lo que decidimos fue implementar la seguridad y que compruebe que sea un BearerToken para ejecutar la acción y tenemos que cuando compruebe como el token te lo devuelve con Bearer = *tokenCompleto….* Aquí algunos ejemplos de llamadas a las Api´s.

Productos

```
get("/booster", { this: OpenApiRoute
    description = "Conseguir los productos con la categoría BOOSTER"
    response { this: OpenApiResponses
        default { this: OpenApiResponse
            description = "Lista con todos los productos con categoría BOOSTER"
        }
        HttpStatusCode.OK to { this: OpenApiResponse
            description = "Lista de los productos"
            body<List<ProductResponseDto>> { description = "Lista de los productos con categoría BOOSTER encontrados" }
        }
        HttpStatusCode.Unauthorized to{ this: OpenApiResponse
            description = "El token no sea válido"
        }
    }
}){ this: PipelineContext<Unit, ApplicationCall>
    val token = call.request.headers["Authorization"]

    val myScope = CoroutineScope(Dispatchers.IO)
    if (token != null){
        val res = myScope.async { client.getAllBoosters(token.toString()) }.await()
        val body = res.body()
        if (res.isSuccessful && body != null){
            call.respond(HttpStatusCode.OK,body)
        }else call.respond(HttpStatusCode.fromValue(res.code()), json.parseToJsonElement(res.errorBody()?.string()!!))
    }else{
        val res = myScope.async { client.getAllBoostersByUser() }.await()
        val body = res.body()
        if (res.isSuccessful && body != null){
            call.respond(HttpStatusCode.OK,body)
        }else{
            call.respond(HttpStatusCode.fromValue(res.code()), json.parseToJsonElement(res.errorBody()?.string()!!))
        }
    }
```

Usuarios

```
fun Application.usuariosRoutes() {
    val client: RetroFitRest by inject(qualifier = named( name: "apiUsuarios"))
    val json = Json { prettyPrint = true }
    routing { this: Routing
        route("/users") { this: Route
            post("/register") { this: PipelineContext<Unit, ApplicationCall>
                val dto = call.receive<UserCreateDto>()
                val res = async(Dispatchers.IO) { this: CoroutineScope
                    client.registerUser(dto)
                }.await()
                val body = res.body()
                if (res.isSuccessful && body != null) {
                    call.respond(HttpStatusCode.Created, body)
                } else
                    call.respond(
                        HttpStatusCode.fromValue(res.code()),
                        json.parseToJsonElement(res.errorBody()?.string()!!)
                    )
            }
```

Pedidos

```
authenticate { this: Route
    post { this: PipelineContext<Unit, ApplicationCall>
        val token = call.request.headers["Authorization"]?.replace( oldValue: "Bearer ", newValue: "").toString()
        val service = call.receive<OrderCreateDto>()
        val myScope = CoroutineScope(Dispatchers.IO)
        val res = myScope.async { client.creteOrder(service, token!!) }.await()
        val body = res.body()
        try {
            if (res.isSuccessful && body != null) {
                call.respond(HttpStatusCode.Created, body)
            }
            } catch (e: OrderBadRequest) {
                call.respond(HttpStatusCode.BadRequest, e.message.toString())
            }
        }
    }
```

TestImplementation(MockKito)

MockKito

MockKito simplemente es implementar la interfaz de testeo Mockito sobre la plataforma de Kotlin al igual que la utilizábamos en Java junto a que ciertas comprobaciones son más amenas y fáciles de aplicar e implementar.

Implementación sobre las rutas

Esquematización Base para cada test: Llamar a lo que se necesita y crear datos de test.

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
class ProductControllerTest {

    @MockK
    private lateinit var tokenService: TokenService

    @MockK
    private lateinit var service: ProductService

    @InjectMockKs
    private lateinit var controller: ProductController

    private val test = Product(
        id = 1, uuid = UUID.randomUUID().toString(), name = "Test", price = 2.50, available = true,
        description = "Prueba descripcion", url = "url", category = ProductCategory.BOOSTER, stock = 10,
        brand = "marca", model = "model"
    )

    private val testDto = ProductDto(
        name = "Test", price = 2.50, available = true,
        description = "Prueba descripcion", url = "url", category = "BOOSTER", stock = 10,
        brand = "marca", model = "model"
    )


    init {
        MockKAnnotations.init( ...obj: this)
    }
```

Productos

```
@Test
fun getAllProducts() = runTest { this: TestScope
    coEvery { service.findAllProducts() } returns listOf(test)
    coEvery { tokenService.getRoles(any()) } returns "ADMIN"
    val all = controller.getAllProducts( token: "eyJhbGciOiJIUzUxMiIsInR
    val result = all.body
    assertAll(
        { assertNotNull(result) },
        { assertTrue(result?.isNotEmpty()!!) },
        { assertEquals(test.uuid, result!![0].uuid) },
        { assertEquals(test.name, result!![0].name) },
        { assertEquals(test.price, result!![0].price) },
        { assertEquals(test.available, result!![0].available) },
        { assertEquals(test.description, result!![0].description) },
        { assertEquals(test.url, result!![0].url) },
        { assertEquals(test.category.name, result!![0].category) },
        { assertEquals(test.stock, result!![0].stock) },
        { assertEquals(test.brand, result!![0].brand) },
        { assertEquals(test.model, result!![0].model) }
    )

    coVerify(exactly = 1) { service.findAllProducts() }
    coVerify(exactly =1) {tokenService.getRoles(any())}
}
```

```
@Test
fun updateProduct() = runTest { this: TestScope
    coEvery { service.findProductByUuid(test.uuid) } returns test
    coEvery { service.updateProduct(any(), any()) } returns test
    coEvery { tokenService.getRoles(any()) } returns "ADMIN"

    val update = controller.updateProduct( token: "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.ey
        testDto, test.uuid)
    val result = update.body

    assertAll(
        { assertNotNull(result) },
        { assertEquals(test.uuid, result!!.uuid) },
        { assertEquals(test.name, result!!.name) },
        { assertEquals(test.price, result!!.price) },
        { assertEquals(test.available, result!!.available) },
        { assertEquals(test.description, result!!.description) },
        { assertEquals(test.url, result!!.url) },
        { assertEquals(test.category.name, result!!.category) },
        { assertEquals(test.stock, result!!.stock) },
        { assertEquals(test.brand, result!!.brand) },
        { assertEquals(test.model, result!!.model) }
    )

    coVerify(exactly = 1) { service.findProductByUuid(test.uuid) }
    coVerify(exactly = 1) { service.updateProduct(any(), any()) }
    coVerify(exactly = 1) { tokenService.getRoles(any()) }
}
```

Pedidos

```
@Test
@org.junit.jupiter.api.Order(1)
fun testGetAll() = testApplication { this: ApplicationTestBuilder
    environment { config }
    val response = client.get("/Orders")
    assertEquals(HttpStatusCode.OK, response.status)
}
@Test
@org.junit.jupiter.api.Order(2)
fun testPost() = testApplication { this: ApplicationTestBuilder
    environment { config }

    val client = createClient { this: HttpClientConfig<out HttpClientEngineConfig>
        install(ContentNegotiation){ this: ContentNegotiation.Config
            json()
        }
    }

    val response = client.post("/pedidos"){ this: HttpRequestBuilder
        contentType(ContentType.Application.Json)
        setBody(create)
    }
    println(response.bodyAsText())
    val result = response.bodyAsText()
    val dept = json.decodeFromString<OrderDto>(result)
    assertAll(
        {assertEquals(order.price, dept.price)}
    )
}
```

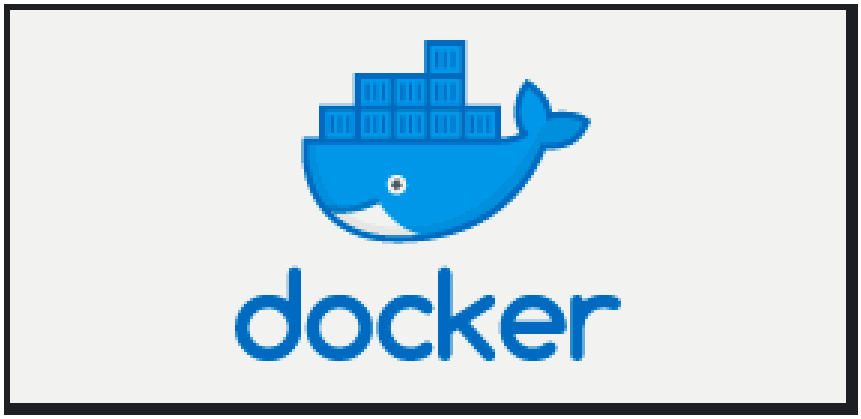
```
@Test
@org.junit.jupiter.api.Order(3)
fun testPatchNotFound() = testApplication { this: ApplicationTestBuilder
    environment { config }
    val client = createClient { this: HttpClientConfig<out HttpClientEngineConfig>
        install(ContentNegotiation){ this: ContentNegotiation.Config
            json()
        }
    }


    val response = client.patch("/pedidos/${UUID.randomUUID()}"){ this: HttpRequestBuilder
        contentType(ContentType.Application.Json)
        setBody(create)
    }

    assertEquals(HttpStatusCode.BadRequest, response.status)
}
```

Servicios

```
@Test
fun findAll() = runTest { this: TestScope
    coEvery { service.findAllServices() } returns listOf(serviceTest)
    coEvery { tokenService.getRoles(any()) } returns "ADMIN"

    val result = controller.getAllServices(tokenSuperadmin)
    val res = result.body!!

    assertAll(
        { assertEquals(result.statusCode, HttpStatus.OK) },
        { assertEquals(res, listOf(serviceTest)) }
    )
    coVerify { service.findAllServices() }
    coVerify { tokenService.getRoles(any()) }
}
```

```
@Test
fun create() = runTest { this: TestScope
    coEvery { service.saveService(any()) } returns serviceTest
    coEvery { tokenService.getRoles(any()) } returns "SUPERADMIN"

    val result = controller.saveService(
        tokenSuperadmin,
        ServiceCreateDto(
            serviceTest.category.name,
            serviceTest.description,
            serviceTest.price,
            serviceTest.url,
        )
    )
    val res = result.body!!

    assertAll(
        { assertEquals(result.statusCode, HttpStatus.CREATED) },
        { assertEquals(res.price, serviceTest.price) },
        { assertEquals(serviceTest.category, res.category) },
        { assertEquals(serviceTest.url, res.url) },
        { assertEquals(serviceTest.available, res.available) }
    )
    coVerify { service.saveService(any()) }
    coVerify { tokenService.getRoles(any()) }
}
```

Docker



Algunas de las razones que hemos decidido en utilizar Docker es que cuenta con una posibilidad de aislamiento y portabilidad muy grande esto significa que los microservicios podemos ejecutarlo en un ambiente diferente y los contenedores podemos subirlos a internet para mayor portabilidad. Otro aspecto a destacar es la escalabilidad a que nos referimos con eso a que de forma rápida y sencilla sobre cada microservicio podemos agregar o eliminar instancias de algún servicio como por ejemplo cambiar de base de datos agregar otro cliente de base de datos etc. Además nos permite crear imágenes con todas las dependencias lo cual nos quita la incertidumbre de saber si todos los servicios utilizan o no las dependencias que necesitan.

Ejemplo de Estructura Docker Compose con la Api General y las que cuelgan de la misma.

```yaml
api-general:
  container_name: api-general
  build: ./apiGeneral
  ports:
    - "8080:8080"
  depends_on:
    - servicio-productos
    - servicio-usuarios
    - servicio-pedidos
  networks:
    - red
```

```yaml
mongo-express:
  image: mongo-express
  container_name: mongo-express
  ports:
    - "8084:8084"
  networks:
    - mongo-network
  depends_on:
    - mongodb-server
  environment:
    ME_CONFIG_MONGODB_ADMINUSERNAME: mongoadmin
    ME_CONFIG_MONGODB_ADMINPASSWORD: mongopass
    ME_CONFIG_MONGODB_SERVER: mongodb-server
  restart: unless-stopped
```

```yaml
mongodb-server:
  image: mongo
  container_name: mongodb-server

  ports:
    - "27017:27017"
  environment:
    MONGO_INITDB_ROOT_USERNAME: mongoadmin
    MONGO_INITDB_ROOT_PASSWORD: mongopass
    MONGO_INITDB_DATABASE: tiendaMusica
  command: --auth
  volumes:
    - ./init:/docker-entrypoint-initdb.d
    - mongo-vol:/data/db
  networks:
    - mongo-network
  restart: always
```

T

```yaml
mariaDb:
  image: mariadb:latest
  container_name: mariaDb
  extends:
    file: ./ApiProducto/docker/docker-compose.yml
    service: mariaDb
  networks:
    - red
```

Logger

Logger es un mecanismo de muestreo de resultados por consola de forma sistemática el cual nos brinda de varias opciones de muestreo de los datos dependiendo del uso que le queremos dar a lo que enseñemos por pantalla. Tales como:

- `logger.trace { "This is trace log" }`
- `logger.debug { "This is debug log" }`
- `logger.info { "This is info log" }`
- `logger.warn { "This is warn log" }`
- `logger.error { "This is error log" }`

Para más información e implementación y configuración de la misma podéis visitar el link que nos redirige a la página Baeldung de Kotlin la cual nos especifica paso por paso lo que hemos de hacer. → CLIC AQUI

Lenguaje

Hemos debatido sobre que lenguaje hemos de utilizar o de hacer nuestro proyecto con una arquitectura en multilenguaje la cual nos tomaría tiempo, al final lo que hemos decidido es utilizar Kotlin ya que el mismo frente a otros como JAVA está más optimizado en la realización de clases, métodos, tecnologías, ciertas librerías de procesamiento de datos tales como la descrita anteriormente la cual era Kotlin Serialization que frente a JAVA no se encuentra. Y es un lenguaje el cual cuenta con características superiores a java como la opción de no admitir tipos sin formato o la seguridad a la hora de trabajar con la Nullability que es el manejo de nulos de java. Por esto nos pareció una buena opción. :D

Referencias

https://www.baeldung.com/kotlin/

https://github.com/joseluisgs?tab=repositories

https://docs.docker.com

https://es.wikipedia.org

https://r2dbc.io