UT 3. Comunicación con el exterior. Ktor y JWT.

1. Introducción

La seguridad es un pilar fundamental en el desarrollo y mantenimiento de software. Al exponer servicios, ya sean REST o de otra índole, es imperativo garantizar la identidad del usuario (autenticación) y verificar que posea los permisos necesarios para acceder al recurso (autorización). Tras evaluar soluciones personalizadas que no cumplieron con las expectativas de robustez, actualmente se emplean protocolos estándar de la industria como OAuth y JWT

2. Evolución histórica

HTTP se ha convertido en uno de los protocolos más utilizados para la comunicación entre aplicaciones, desde sus inicios se ha intentado garantizar la seguridad de acceso, las evolución de estas técnicas ha sido:

La Era de la "Inexistencia" (Protocolo Stateless)

En los inicios de la web, el protocolo HTTP era puramente sin estado (stateless). Cada petición era independiente y no había forma de saber si el usuario que pedía la página A era el mismo que pedía la página B.

Mecanismo: El acceso era público o se limitaba por direcciones IP.

Abandono: No permitía la personalización ni el comercio electrónico.

Autenticación Básica y Cookies (Lado del Cliente)

Para resolver la falta de estado, se introdujeron las cookies y la autenticación básica (Authorization: Basic).

Desventajas (Abandono):

- Inseguridad: Las credenciales se enviaban en cada petición codificadas en Base64 (fáciles de interceptar sin SSL).
- Vulnerabilidad: Muy susceptible a ataques de fijación de sesión.

Sesiones en el Servidor (Stateful)

Se popularizó el uso de un Session ID almacenado en una cookie, mientras que los datos del usuario se guardaban en la memoria del servidor.

Mecanismo: El servidor crea un archivo/registro para el usuario y le entrega una "llave" (ID).

Desventajas (Abandono parcial en APIs modernas):

- Escalabilidad: Si tienes 10 servidores, todos deben compartir la misma base de datos de sesiones (Sticky Sessions).
- Consumo de recursos: El servidor debe mantener miles de sesiones activas en RAM.

Certificados SSL/TLS en Cliente y Servidor (mTLS)

Aunque el SSL/TLS es estándar para cifrar el canal, el uso de certificados de cliente es una forma de validación donde el navegador presenta un certificado digital para identificarse.

Estatus: Muy usado en entornos gubernamentales, militares o bancarios.

Desventajas:

- Complejidad: Es difícil de gestionar para el usuario común (instalar archivos .p12 o .pfx).
- Rigidez: El acceso queda ligado a un dispositivo específico.

La Revolución de OAuth y JWT (Modelos Actuales)

Con la llegada de las arquitecturas de microservicios y aplicaciones móviles, el control se movió hacia los Tokens.

OAuth 2.0 (Delegación de Autorización)

Permite que una aplicación acceda a recursos en nombre de un usuario sin conocer su contraseña (ej. "Iniciar sesión con Google").

Ventaja: Permite revocar accesos de aplicaciones específicas de forma independiente.

JWT (JSON Web Tokens)

Es un estándar para transmitir información de forma segura entre partes como un objeto JSON.

Mecanismo: El token es "autocontenido"; lleva dentro la información del usuario y está firmado digitalmente.

Por qué domina hoy:

Sin Estado (Stateless): El servidor no necesita guardar nada en memoria; solo verifica la firma del token.

Escalabilidad Total: Ideal para apps multiplataforma y sistemas distribuidos.

3. Estándar JWT

El estándar **JWT**, cuya documentación y herramientas de prueba se encuentran en jwt.io, define un formato compacto y autónomo para transmitir información como un objeto JSON. Técnicamente, un JWT es una cadena de caracteres estructurada que permite almacenar información variada de forma segura. Su arquitectura es comparable a la de un paquete en la capa de aplicación del modelo TCP/IP, dividiéndose en tres partes fundamentales:

Cabecera (Header)

Típicamente consta de dos partes: el tipo de token (JWT) y el algoritmo de firma utilizado, como **HMAC SHA256** o **RSA**. Esta sección indica al receptor cómo debe interpretar y validar el token.

Carga útil (Payload)

Contiene las **reclamaciones** (*claims*), que son declaraciones sobre una entidad (generalmente el usuario) y metadatos adicionales. Según el estándar RFC 7519, existen tres categorías:

Reclamaciones Registradas: Son un conjunto de campos predefinidos, no obligatorios pero recomendados para garantizar la interoperabilidad. Ejemplos comunes incluyen:

iss (Emisor): Quien genera el token.

sub (Sujeto): El ID del usuario.

exp (Expiración): Fecha y hora de caducidad.

aud (Audiencia): El destinatario del token.

Reclamaciones Públicas: Definidas a voluntad por los desarrolladores. Para evitar colisiones de nombres, se recomienda que sigan el Registro de Tokens Web de la IANA o utilicen nombres únicos (como una URL).

Reclamaciones Privadas: Son campos personalizados diseñados para compartir información específica entre las partes que acuerdan su uso (por ejemplo, el rol del usuario en la aplicación).

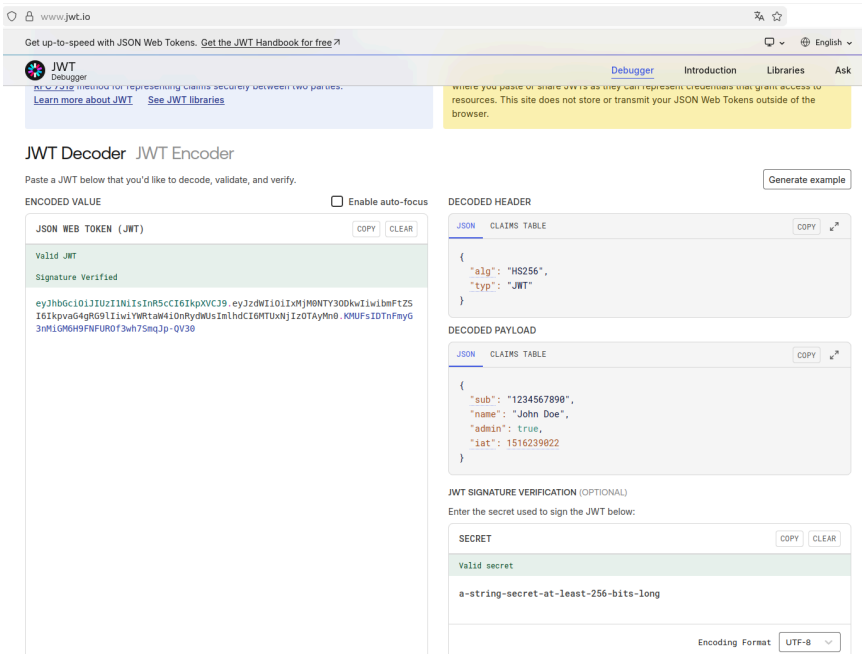
Firma (Signature)

Es la parte crítica que garantiza la integridad del token. Se utiliza para verificar que el mensaje no fue alterado durante la transmisión y, en el caso de usar claves privadas, confirma la identidad del remitente.

Para calcularla, se toma la **cabecera** y la **carga útil** (ambas codificadas en Base64Url), se concatenan con un punto y se procesan mediante el algoritmo especificado (como HMAC SHA256) junto con una clave secreta o contraseña.

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

En la página web https://www.jwt.io/ se puede comprobar el funcionamiento de JWT, al pasarle un token descodifica la información y muestra cada uno de las partes, pudiendo comprobar si firma es la correcta insertando la clave.



Observar la separación de las partes en los puntos del token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiO
iIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWR
taW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0.KMUFsIDTn
FmyG3nMiGM6H9FNFUROf3wh7SmqJp-QV30
```

Para acceder a un recurso protegido que requiere autenticación, el token debe enviarse en cada petición. El estándar establece que debe incluirse en la cabecera **Authorization**, utilizando el esquema **Bearer** seguido del valor del token: **Bearer <token>**

Aunque del lado del cliente no es importante, del lado del servidor cuando llega un token se ha de comprobar que el token sea válido en cuanto al formato y verificar que cumple con la autenticidad y la integridad del mismo.

La fase de validación comprueba:

Estructura: Asegurar que el token tenga las tres partes estándar (encabezado, carga útil, firma) separadas por puntos.

Formato : Verificación de que cada parte está codificada correctamente (Base64URL) y que la carga útil contiene las reclamaciones esperadas.

Contenido: Comprobar si las reclamaciones dentro de la carga útil son correctas, como el tiempo de caducidad (exp), emitido en (iat), no antes (nbf), entre otros, **para asegurarse de que el token no haya caducado,** no se use antes de su tiempo, etc.

La fase de verificación se encarga de:

Verificación de firma: Este es el aspecto principal de la verificación donde la parte de firma de la JWT se comprueba con la cabecera y la carga útil. Esto se

hace utilizando el algoritmo especificado en el encabezado (como HMAC, RSA o ECDSA) con una clave secreta o clave pública. Si la firma no coincide con lo que se espera, el token podría haber sido manipulado o no de una fuente confiable.

Verificación del emisor: Comprobar si la reclamación coincide con un emisor esperado.

Verificación de audiencia: Asegurarse de que la reclamación de aud coincida con la audiencia esperada.

4. Ciclo de vida

Los token pasan por diferentes fases desde que se crean hasta que se destruyen, las etapas por las que pasa son:

Creación (Issuance): Ocurre tras el login exitoso. El servidor genera el JWT y lo firma.

Validación (Verification): En cada petición, el servidor comprueba la firma y la fecha de expiración (exp).

Renovación (Refresh): Proceso de obtener un nuevo token de acceso antes de que el actual expire.

Existen dos alternativas a la hora de gestionar el ciclo de vida de los tokens:

Etapas con un token (access token)

Autenticación: El usuario envía sus credenciales.

Emisión: El servidor genera un JWT con un tiempo de vida (expiración) determinado.

Uso: El cliente envía el token en cada petición.

Finalización: Una vez que el token expira, el servidor devuelve un 401 Unauthorized y el cliente obliga al usuario a loguearse de nuevo para obtener uno nuevo. No hay renovación automática en segundo plano.

Etapas con dos token (access token y refresh token)

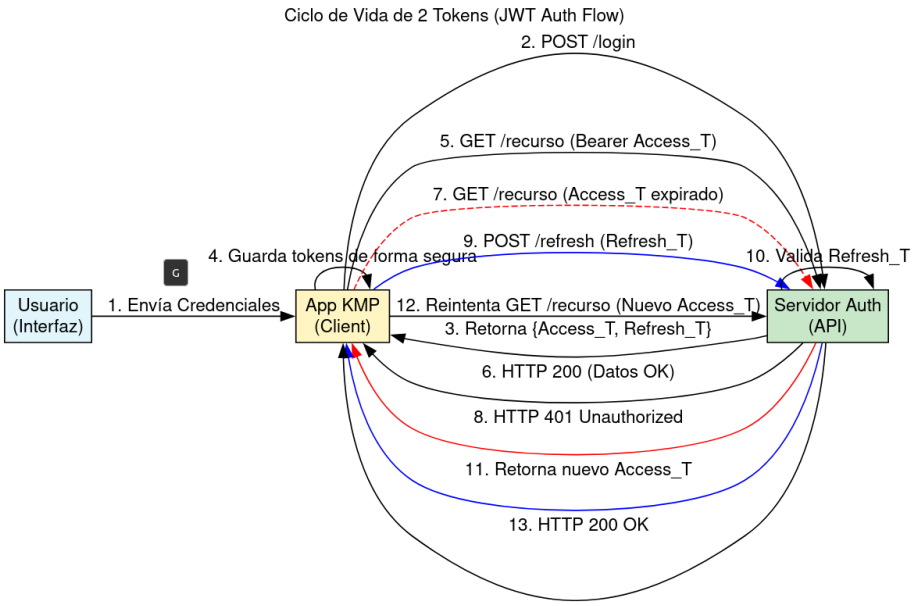
Emisión (Sign-in): El servidor genera un Access Token (vida corta, ej. 15 min) y un Refresh Token (vida larga, ej. 7 días).

Uso (Request): El cliente envía el Access Token en el header Authorization: Bearer <token>.

Expiración (Expiration): El servidor responde con un error 401 Unauthorized cuando el Access Token caduca.

Renovación (Rotation): El cliente detecta el 401 y envía el Refresh Token a un endpoint especial para recibir un nuevo par de tokens sin intervención del usuario.

Revocación (Logout): El ciclo termina cuando los tokens se eliminan del cliente o se invalidan en una "lista negra" (blacklist) en el servidor (recomendado pero no obligatorio, excepto en aplicaciones bancarias, de salud o cualquier sistema que maneje datos sensibles.)



5. Ktor y JWT en el servidor

Ktor como la mayoría de las librerías/framework que trabajan con HTTP y servicios web (REST) posee un plugin para trabajar con JWT, para:

Validación de Integridad y Firma.

La función más crítica es verificar que el token no haya sido manipulado. Ktor hace esto automáticamente al:

Verificar la firma: Utiliza una clave secreta (HMAC) o una clave pública (RSA/ECDSA) para confirmar que el token fue emitido por tu servidor.

Comprobar la expiración: Revisa el campo exp (expiration) y rechaza el token si ha caducado.

Validar Emisor y Audiencia: Confirma que el token viene de donde dice (iss) y es para quien dice (aud).

Extracción y Verificación de "Claims".

Ktor permite inspeccionar el contenido del token (Payload) para tomar decisiones de lógica de negocio:

Identificación del usuario: Extrae el sub (Subject) o el ID del usuario para saber quién hace la petición.

Validación personalizada: Añadir reglas extra, por ejemplo: "Solo permitir el paso si el token tiene el claim 'email_verified' como true".

Gestión de Roles y Permisos (RBAC)

Se puede utilizar la información dentro del JWT para restringir el acceso a ciertas partes de tu API:

- Autorización: Verificar si el usuario tiene el rol de "ADMIN" o "PREMIUM".

Scopes: Limitar qué puede hacer el cliente (ej. read:profile, write:settings).

La generación de los tokens no es misión de jwt-auth pudiendo usar otras librerías para su creación, la más utilizada es com.auth0.jwt (conocida como Java-JWT) es la implementación de referencia para trabajar con JSON Web Tokens en el ecosistema Java/Kotlin.

5.1. Generar token

La generación del token se realiza usand la librería com.auth0.jwt, se incluye junto con la librería de Ktor-auth-jwt, encargado de la validación y la verificación.

Fichero libs.versions.toml:

```
ktor-server-auth = { module = "io.ktor:ktor-server-auth",
version.ref = "ktor" }
ktor-server-auth-jwt = { module = "io.ktor:ktor-server-auth-jwt",
version.ref = "ktor" }
```

Fichero build.gradle.kts:

```
implementation(libs.ktor.server.auth)
implementation(libs.ktor.server.auth.jwt)
```

La generación de un token es sencilla, se usa el método estático JWT.create, al que se le van añadiendo elementos de la cadena token vistos anteriormente como cabeceras, reclamaciones o firma.

Reclamaciones registradas.

Poseen métodos propios para indicar el valor en el token las principales son:

Claim	Nombre	Función en com.auth0.jwt	Descripción
sub	Subject	.withSubject(String)	Identifica al usuario (ID o Email).
exp	Expiration	.withExpiresAt(Date)	Fecha/hora de caducidad (obligatorio por seguridad).
iss	Issuer	.withIssuer(String)	Quién emite el token (tu dominio o nombre de app).
aud	Audience	.withAudience(String)	Para quién es el token (tu API o cliente específico).

Claim	Nombre	Función en com.auth0.jwt	Descripción
iat	Issued At	.withIssuedAt(Date)	Cuándo se creó (útil para auditoría).
nbf	Not Before	.withNotBefore(Date)	El token no será válido antes de esta fecha.
jti	JWT ID	.withJWTId(String)	ID único del token (para listas negras).

Las cuatro primeras se pueden considerar imprescindibles.

Reclamaciones públicas y privadas, se utiliza el método withClaim(nombre-chaim, role):

```
.withClaim("type", "refresh")
.withClaim("role", role)
```

También existe un método que indica el algoritimo de firma (encabezamiento) y la clave de la firma.

```
.sign(Algorithm.HMAC256("secret"))
```

Un ejemplo de generación de token:

```
val accessToken= JWT.create()
     .withSubject(email)
     .withIssuer("https://jwt-provider-domain/")
     .withAudience("jwt-audience")
     .withClaim("type", "refresh")
     .withClaim("role", role)
     .withExpiresAt(Date(System.currentTimeMillis() + 3600000)) //
1 hora
     .sign(Algorithm.HMAC256("secret"))
```

Se recomienda crear un servicio en la capa de infraestructura que genere los tokens y definirlo como singleton para inyectarlo donde sea necesario.

5.2. Configuración de Ktor. Validación y verificación

Al igual que otras funcionalidades de Ktor, la autenticación debe instalarse y configurarse como un plugin. El plugin auth es versátil y permite implementar diversos métodos, como JWT (JSON Web Tokens) o Cookies.

Esta configuración es fundamental, ya que establece los requisitos de seguridad que debe cumplir el token recibido, definiendo tanto el mecanismo de verificación (integridad y origen) como el de validación (lógica de negocio y permisos).

Ejemplo: Configuración Multitenant (Usuario y Administrador)

En el siguiente ejemplo, se definen dos estrategias de autenticación independientes (denominadas schemes). Cada una posee su propio **Realm (un identificador del área protegida)** y se divide en dos fases críticas:

Fase de Verificación (verifier): Ktor comprueba automáticamente la estructura del token (que tenga sus tres partes), la validez de la firma criptográfica, el emisor (iss) y la fecha de expiración (exp). Si alguna de estas pruebas falla, Ktor detiene la petición inmediatamente con un error 401 Unauthorized, sin llegar a ejecutar la lógica de tu ruta.

Fase de Validación (validate): Una vez que el token es auténtico, entra en esta sección para comprobar el contenido personalizado (claims). Aquí es donde se decide si, por ejemplo, el usuario tiene el rol adecuado para acceder.

```
fun Application.configureSecurity() {
    authentication {
        // Configuración para la API normal del usuario
        jwt("auth-user") {
            realm = "Access to user api"
            verifier(
                JWT.require(Algorithm.HMAC256("secreto-1"))
                    .withIssuer("mi-app.com")
                    .build()
            )
            validate { credential ->
                if (credential.payload.getClaim("role").asString()
≠ null) {
                    JWTPrincipal(credential.payload)
                } else null
            }
        }

        // Configuración esspecífica para Administradores (con
otra clave o reglas)
        jwt("auth-admin") {
            realm = "Access to admin panel"
            verifier(

JWT.require(Algorithm.HMAC256("secreto-admin-ultra-seguro"))
                    .withIssuer("admin.mi-app.com")
                    .build()
            )
            validate { credential ->
                val role =
```

```
credential.payload.getClaim("role").asString()
                if (role == "ADMIN") {
                    JWTPrincipal(credential.payload)
                } else null // Si no es admin, falla aunque el
token sea válido
            }
        }
    }
}
```

En caso de que la validación o la verificación no tengan éxito, Ktor interrumpe el flujo de la petición y devuelve automáticamente una respuesta HTTP 401 Unauthorized.

Como se puede observar, existen valores (como el secreto, la audiencia o el emisor) que deben coincidir tanto en la generación como en la verificación del token. Para evitar la duplicación de código y errores manuales, Ktor permite centralizar estos datos en un fichero de configuración ubicado en la carpeta resources (generalmente application.conf en formato HOCON, aunque también soporta YAML o Properties) , por ejemplo para definir la configuración anterior:

```
jwt {
    user {
        secret = "secreto-1"
        issuer = "mi-app.com"
        realm = "Access to user api"
    }
    admin {
        secret = "secreto-admin-ultra-seguro"
        issuer = "admin.mi-app.com"
        realm = "Access to admin panel"
    }
}
```

Para consumir estos valores en el plugin de autenticación, accedemos al objeto environment.config. Es importante notar que el verifier se encarga de la integridad técnica (firma y formato), mientras que el bloque validate se encarga de la lógica de negocio (roles).

```
fun Application.configureSecurity() {
    // Extracción de parámetros para Usuario
    val userSecret =
environment.config.property("jwt.user.secret").getString()
    val userIssuer =
environment.config.property("jwt.user.issuer").getString()
    val userRealm =
environment.config.property("jwt.user.realm").getString()
```

```
    // Extracción de parámetros para Administrador
    val adminSecret =
environment.config.property("jwt.admin.secret").getString()
    val adminIssuer =
environment.config.property("jwt.admin.issuer").getString()
    val adminRealm =
environment.config.property("jwt.admin.realm").getString()

    authentication {

        jwt("auth-user") {
            realm = userRealm
            verifier(
                JWT.require(Algorithm.HMAC256(userSecret))
                    .withIssuer(userIssuer)
                    .build()
            )
            validate { credential ->
                if (credential.payload.getClaim("role").asString()
!= null) {
                    JWTPrincipal(credential.payload)
                } else null
            }
        }

        jwt("auth-admin") {
            realm = adminRealm
            verifier(
                JWT.require(Algorithm.HMAC256(adminSecret))
                    .withIssuer(adminIssuer)
                    .build()
            )
            validate { credential ->
                val role =
credential.payload.getClaim("role").asString()
                if (role == "ADMIN") {
                    JWTPrincipal(credential.payload)
                } else null
            }
        }
    }
}
```

Y para la generación del token:

```
routing {
    post("/login") {
```

```
        // Extraer datos del fichero application.conf
        //
        val config = call.application.environment.config
        val secret =
config.property("jwt.user.secret").getString()
        val issuer =
config.property("jwt.user.issuer").getString()
        val audience =
config.property("jwt.user.audience").getString()

        // Datos del usuario
        val email = "usuario@ejemplo.com"
        val role = "USER"

        // Generación del Token usando los valores del fichero
        val accessToken = JWT.create()
            .withSubject(email)
            .withIssuer(issuer)      //  Valor del fichero
            .withAudience(audience)  // Valor del fichero
            .withClaim("type", "access")
            .withClaim("role", role)
            .withExpiresAt(Date(System.currentTimeMillis() +
3600000)) // 1 hora
            .sign(Algorithm.HMAC256(secret)) // Secreto del
fichero

        call.respond(mapOf("token" to accessToken))
    }
}
```

En el caso de ser un servicio de la capa de infraestructura (generación de token) se ha de pasar a este servicio los valores, por ejemplo, inyectándolos con Koin:

```
data class JwtSettings(
    val secret: String,
    val issuer: String,
    val audience: String
)
```

Proveedor/servicio de infraestructura, encargado de crear los tokens:

```
class JwtTokenProvider(private val settings: JwtSettings) {

    fun generateAccessToken(email: String, role: String): String {
        return JWT.create()
            .withSubject(email)
            .withIssuer(settings.issuer)
```

```
            .withAudience(settings.audience)
            .withClaim("role", role)
            .withClaim("type", "access")
            .withExpiresAt(Date(System.currentTimeMillis() +
3600000)) // 1 hora
            .sign(Algorithm.HMAC256(settings.secret))
    }
}
```

Configuración en Koin:

```
single {
    val config = get<Application>().environment.config //el plugin
de Koin para Ktor registra automáticamente la instancia actual de
la Application en su contenedor.
    JwtSettings(
        secret = config.property("jwt.user.secret").getString(),
        issuer = config.property("jwt.user.issuer").getString(),
        audience =
config.property("jwt.user.audience").getString()
    )
}

// Inyectar JwtSettings
single { JwtTokenProvider(get()) }
```

5.3. Accediendo a la información del token

Una vez que el token ha pasado el filtro de autenticación de Ktor, la información se almacena en un objeto llamado Principal. Ktor extrae automáticamente los datos del token y los adjunta al contexto de la llamada (call).

Para que los casos de Uso (capa de Aplicación) puedan usar esta información sin depender de Ktor, el flujo estándar es extraer los datos en la ruta y pasarlos como argumentos.

```
 jwt("auth-admin") {
            realm = adminRealm
            verifier(
                JWT.require(Algorithm.HMAC256(adminSecret))
                    .withIssuer(adminIssuer)
                    .build()
            )
            validate { credential →
                val role =
credential.payload.getClaim("role").asString()
                if (role == "ADMIN") {
                    JWTPrincipal(credential.payload)
```

```
                } else null
            }
        }
```

Dentro de una ruta protegida por authenticate, usar call.principal<JWTPrincipal>(). Este objeto contiene el Payload original del token.

```
authenticate("auth-user") {
    get("/me/profile") {
        // Extraemos el
        val principal = call.principal<JWTPrincipal>()

        //  Accedder a los claims (Subject, Role, etc.)
        val email = principal?.payload?.subject // El 'sub' que
configuramos
        val role =
principal?.payload?.getClaim("role")?.asString()
        val userId =
principal?.payload?.getClaim("user_id")?.asInt()

        //  Llamar al caso de uUso pasándole  los datos necesarios
        if (email ≠ null) {
            val userProfile = getProfileUseCase.execute(email)
            call.respond(userProfile)
        } else {
            call.respond(HttpStatusCode.Unauthorized)
        }
    }
}
```

5.4. Estandarización de respuestas

Si bien se puede enviar directamente el token una vez creado directamente en el cuerpo del mensaje, existen algunos formatos que se han convertido en estándares de la industria Existen 3 formatos principales.

Estándar OAuth 2.0 (El clásico de la industria)

Es el formato que utilizan la mayoría de las APIs (incluyendo plataformas como GitHub o Slack). Su característica principal es que el access_token es opaco o un JWT, pero siempre acompaña el tiempo de vida en segundos.

```
{
    "access_token": "ya29.a0...",
    "expires_in": 3599,
    "token_type": "Bearer",
```

```
    "refresh_token": "1//06..."
}
```

Estándar Google / OpenID Connect (OIDC)

Google utiliza una extensión de OAuth 2.0. La gran diferencia es que introduce el id_token.

```
{
    "access_token": "ya29.a0...",
    "id_token": "eyJhbG... (JWT con info del perfil)",
    "expires_in": 3599,
    "token_type": "Bearer",
    "refresh_token": "1//06..."
}
```

Diferencia clave: El access_token sirve para llamar a la API (permisos). El id_token es un JWT que contiene los datos del usuario (nombre, foto, email) para que la App cliente pueda mostrarlos sin hacer otra petición extra.

Estándar "Wrapped JWT" o Envoltorio Simple (Estilo Moderno/Firebase)

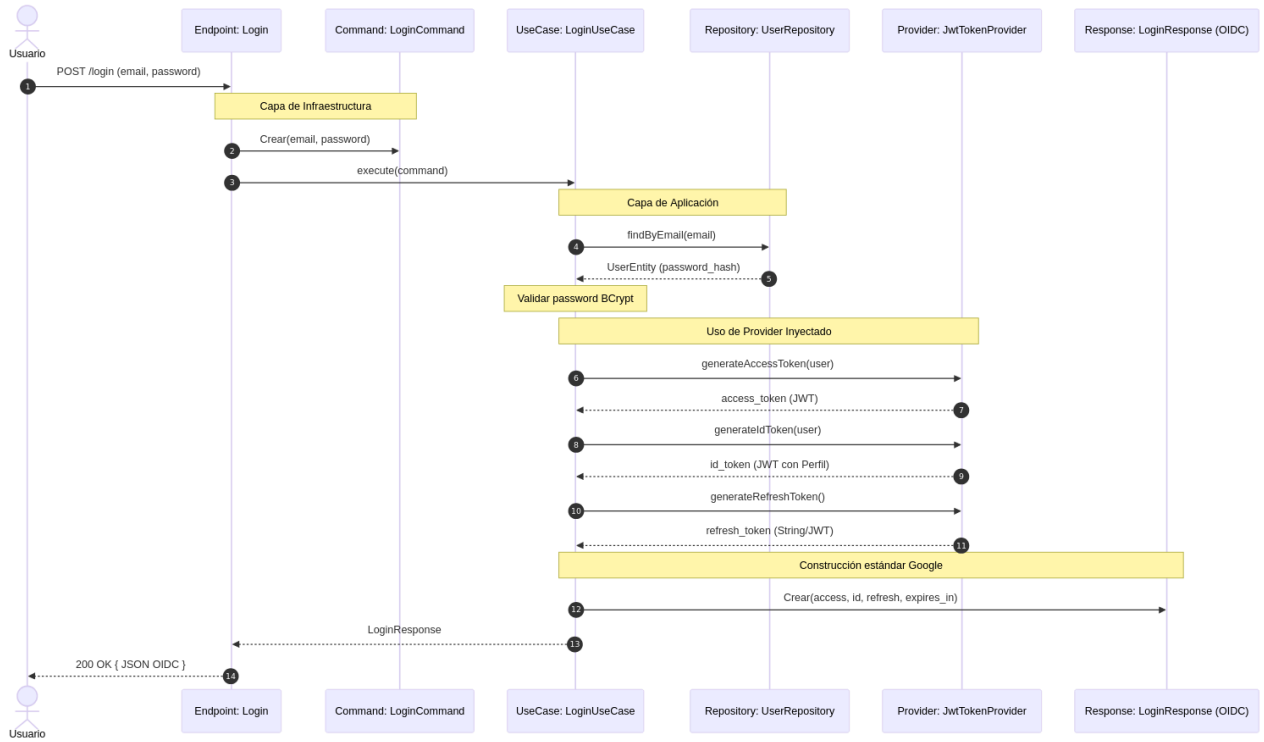
Es común en aplicaciones móviles modernas y microservicios que quieren evitar el estado en el servidor. En lugar de un refresh_token que es una cadena aleatoria, el servidor devuelve un objeto donde ambos son JWTs.

```
{
    "user": {
    "id": "123",
    "email": "user@mail.com"},
    "tokens": {
    "access": "eyJ...",
    "refresh": "eyJ..."}
}
```

Ventaja: El servidor no necesita base de datos para validar el refresh_token (porque es un JWT firmado).

Riesgo: Si el refresh_token es un JWT, no se puede revocar fácilmente a menos que uses una "Blacklist" o listas de revocación.

Posible diagrama de secuencia:



5.5. Validación de rutas

La validación de un endpoint en Ktor es sencilla. Basta con envolver la ruta en un bloque authenticate, indicando el área protegida (configuración de JWT) a la que pertenece. De este modo, la ruta solo se ejecutará si se recibe un token válido que cumpla con los requisitos definidos:

```
authenticate("auth-user") {
    get("/api/profile") {
        val principal = call.principal<JWTPrincipal>()
        val userId =
principal?.payload?.getClaim("user_id")?.asString()
        call.respondText("Hola, usuario $userId")
    }
}
```

Aunque el sistema anterior garantiza que el usuario es quien dice ser (Autenticación), a veces es insuficiente. Por ejemplo, cuando se necesita restringir el acceso según el **rol** del usuario (Autorización).

Una opción es realizar una validación manual dentro de la ruta, pero esto resulta intrusivo, genera duplicidad de código y es poco flexible ante cambios complejos:

```
authenticate("auth-user") {
    get("/api/admin/config") {
       val principal = call.principal<JWTPrincipal>()
```

```
        val role =
principal?.payload?.getClaim("role")?.asString()

        if (role == "ADMIN") {
            call.respond(HttpStatusCode.OK, "Configuración
secreta")
        } else {
            // Retornar 403 Forbidden (Se sabe quién eres, pero no
puedes entrar)
            call.respond(HttpStatusCode.Forbidden, "No tienes
permisos de administrador")
        }
….
    }
}
```

Para solucionar esto de forma elegante, recordar que Ktor se basa en un **Pipeline** (tubería) de ejecución. Se puede insertar lógica personalizada en fases específicas del ciclo de vida de una petición:

Fase (Phase)	**Responsabilidad**	**Relación con JWT / Seguridad**
Setup	Configuración inicial de la llamada.	Prepara el contexto para la autenticación.
Monitoring	Registro (Logging) y métricas.	Ideal para auditar intentos de acceso.
Plugins	Ejecución de la mayoría de plugins instalados.	Aquí se procesan plugins de serialización (JSON).
Authentication	Validación de credenciales.	Aquí actúa el plugin de JWT. Verifica la firma y crea el Principal.
Call	Ejecución de la lógica de tu ruta.	Se accede al call.principal<JWTPrincipal>().
Fallback	Gestión si no se encontró la ruta (404).	Maneja respuestas si el endpoint no existe.

Crear un plugin que se ejecute en la fase de autenticación permite centralizar la lógica, mejorando el mantenimiento. En el siguiente ejemplo, el plugin extrae el *claim* de rol y lo compara con una lista de roles autorizados definida en la configuración:

```
class RoleConfig {
    var requiredRoles: List<String> = emptyList()
```

```
}
val RoleBasedAuthorization = createRouteScopedPlugin(
    name = "RoleBasedAuthorization",
    createConfiguration = ::RoleConfig
) {
    // se engancha en la fase de authenticación
    on(AuthenticationChecked) { call →
        val principal = call.principal<JWTPrincipal>()

        if (principal == null) {
            // Si entra aquí, Ktor no validó el token de Apidog
(Auth falló antes)
            println("DEBUG: No se encontró Principal. Revisa el
bloque authenticate().")
            call.respond(HttpStatusCode.Unauthorized, "No
autenticado")
            return@on
        }

        val userRole =
principal.payload.getClaim("role")?.asString()
        println("DEBUG: Rol encontrado en el token: $userRole")

        if (userRole == null || userRole !in
pluginConfig.requiredRoles) {
            call.respond(HttpStatusCode.Forbidden, "Permisos
insuficientes. Se requiere: ${pluginConfig.requiredRoles}")
        }
    }
}
```

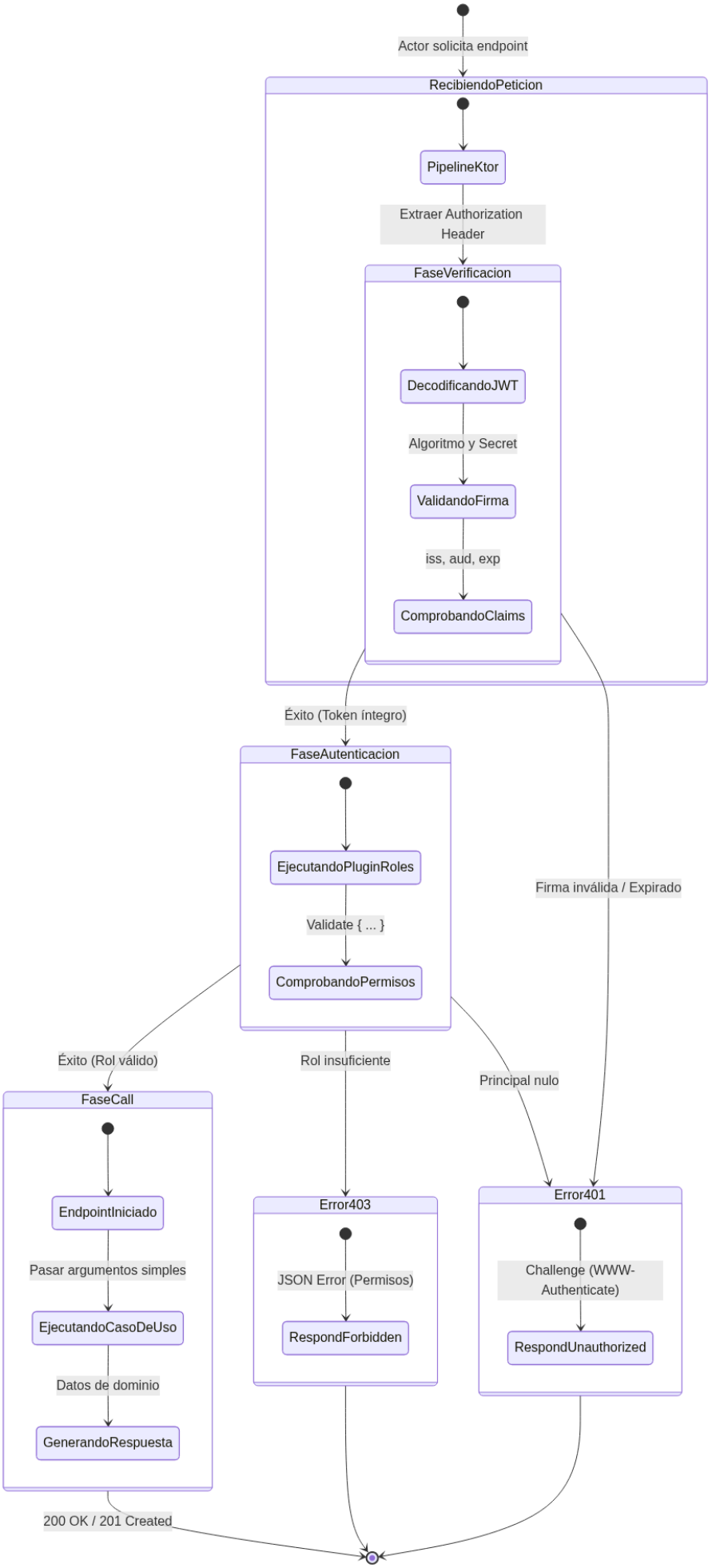
Gracias a este plugin, se puede proteger bloques enteros de rutas indicando los roles permitidos. En este caso, todas las rutas de administración quedan restringidas al rol ADMIN:

```
fun Route.configureAdminRoutes() {

    route("/admin") {
        //solo se permite con auth-jwt token
        authenticate("auth-jwt") {
            install(RoleBasedAuthorization) {
                requiredRoles = listOf(UserRole.ADMIN.toString())
            }
            configureCategoriasRoutes()
            configureDependientesRoutes()
```

```
            configureProductosRoutes()
            configurePedidosRoutes()
        }
    }
}
```

Diagrama de estados para ejecutar un endpoint con seguridad:


Actor solicita endpoint
RecibiendoPeticion
PipelineKtor
Extraer Authorization Header
FaseVerificacion
DecodificandoJWT
Algoritmo y Secret
ValidandoFirma
iss, aud, exp
ComprobandoClaims
Éxito (Token íntegro)
Firma inválida / Expirado
FaseAutenticacion
EjecutandoPluginRoles
Validate { ... }
ComprobandoPermisos
Éxito (Rol válido)
Rol insuficiente
Principal nulo
FaseCall
EndpointIniciado
Pasar argumentos simples
EjecutandoCasoDeUso
Datos de dominio
GenerandoRespuesta
Error403
JSON Error (Permisos)
RespondForbidden
Error401
Challenge (WWW-Authenticate)
RespondUnauthorized
200 OK / 201 Created

6. Ktor y JWT en el cliente

Uno de los principios fundamentales en el desarrollo de sistemas distribuidos es la independencia tecnológica. No resulta necesario que los clientes y el servidor se implementen bajo la misma tecnología o lenguaje de programación; es suficiente con que ambos tengan la capacidad de intercambiar mensajes **HTTP** con contenido en formato **JSON**. Esta estandarización permite que un servidor desarrollado en **Ktor** pueda dar servicio a una amplia gama de clientes, desde aplicaciones web en React hasta sistemas embebidos.

En el ecosistema de Kotlin, **Ktor** se posiciona como una herramienta versátil que puede emplearse tanto en aplicaciones nativas de **Android** como en proyectos de **Kotlin Multiplatform (KMP)**. Esto facilita que el código de red sea compartido entre diferentes plataformas como iOS, Desktop y Web, optimizando los tiempos de desarrollo y mantenimiento.

Aunque la configuración y el uso de Ktor en el lado del cliente suelen ser más sencillos en comparación con la lógica del servidor, el principal desafío técnico reside en la **gestión segura del token**. Esta responsabilidad implica no solo el almacenamiento cifrado de las credenciales en el dispositivo, sino también la implementación de mecanismos de interceptación para la inclusión del encabezado de autorización y la gestión de ciclos de renovación mediante *Refresh Tokens*.

Para garantizar una implementación robusta en aplicaciones cliente, se deben considerar los siguientes pilares:

Persistencia Segura: El almacenamiento de los tokens debe realizarse utilizando mecanismos nativos de seguridad, como el **Keychain** en iOS o **EncryptedSharedPreferences** en Android, gestionados mediante el patrón *expect/actual* de KMP.

Automatización de la Autenticación: El uso del plugin de autenticación de Ktor Client permite automatizar la inserción del esquema **Bearer**, asegurando que cada petición saliente incluya las credenciales necesarias de forma transparente.

Resiliencia ante la Expiración: La lógica del cliente debe ser capaz de procesar respuestas de error **401 Unauthorized**, iniciando automáticamente el flujo de renovación del token y reintentando la petición original sin intervención del usuario.

6.1. Configuración y uso

Para integrar capacidades de autenticación en el cliente de Ktor, se debe incluir el plugin específico en las dependencias del proyecto:

```
implementation("io.ktor:ktor-client-auth:3.3.3")
```

En un escenario de uso simplificado, no es estrictamente obligatorio emplear dicha librería. El flujo básico consiste en realizar la petición de *login*, obtener el token,

almacenarlo temporalmente (por ejemplo, en un objeto *Singleton*) y adjuntarlo manualmente en la cabecera de cada petición que lo requiera:

```
suspend fun getUserProfile(token: String): UserProfile {
    return client.get("https://api.ejemplo.com/profile") {
        // Inyección manual del token Bearer
        header(HttpHeaders.Authorization, "Bearer $token")
    }.body()
}
```

La inyección manual presenta desventajas evidentes, como la duplicidad de código y la dificultad de mantenimiento. Para resolver estos inconvenientes, se utiliza la extensión **Auth**, la cual permite automatizar la inclusión de las credenciales en el encabezado de las peticiones salientes:

```
val client = HttpClient() {
    install(Auth) {
        bearer {
            loadTokens {
                // Configuración básica: se devuelve el token
actual sin lógica de refresco
                 BearerTokens(accessToken = "mi_token_jwt",
refreshToken = "")
            }
        }
    }
}
```

En arquitecturas robustas, el cliente HTTP se define como un componente único (Singleton) gestionado por un contenedor de inyección de dependencias como **Koin**. Esto permite centralizar la configuración de serialización, cabeceras por defecto y seguridad en un solo lugar:

```
single {
    //selecciona el mejor motor
    HttpClient() {
        install(Auth) {
            bearer {
                loadTokens {
                    // Configuración básica: se devuelve el token
actual sin lógica de refresco
                    BearerTokens(accessToken = "mi_token_jwt",
refreshToken = "")
                }
```

```
            }
        }
        install(ContentNegotiation) {
            json(get())
        }
        install(DefaultRequest) {
            header("Content-Type", "application/json")
        }
```

Como se observa en el ejemplo anterior, el uso de una cadena de texto estática ("mi_token_jwt") no es viable en una aplicación real. El token debe ser gestionado dinámicamente: obtenido tras un proceso de autenticación exitoso, almacenado de forma segura en el dispositivo y recuperado por el plugin Auth solo cuando sea necesario.

Además no parece muy seguro enviar el token sea o no necesario, por ejemplo, para acceder a los “endpoints” públicos, pudiendo solucionarlo, analizando la url de petición para incluir o no el token, por ejemplo la aplicación tiene una serie de rutas privadas y otras públicas, las públicas son auth y publicy el resto privadas:

```
bearer {
    // Anyadir credenciales
    loadTokens {
        BearerTokens(accessToken = "mi_token_jwt", refreshToken =
"")
    }

    // Decidir si se añade o no el token en la petición
    sendWithoutRequest { request ->
        // Se extraen los segmentos de la ruta (ej. ["auth",
"login"])
        val path = request.url.pathSegments

        // Se definen las secciones que no requieren autenticación
        val isPublicSection = path.contains("auth") ||
path.contains("public")

        // También es vital asegurar que el host sea el de nuestra
API
        val isExternalHost = request.url.host != "api.mi-app.com"

        // Si es una sección pública o un host externo, se retorna
'true' (No enviar token)
        isPublicSection || isExternalHost
    }
}
```

6.2. Refresco

La renovación de tokens (Token Refresh) es una tarea crítica que Ktor Client puede automatizar mediante su plugin de autenticación. Este mecanismo se activa automáticamente en dos situaciones:

Respuesta del servidor: Cuando se recibe un código 401 Unauthorized.

Validación preventiva: Cuando el plugin detecta que el token almacenado ha caducado o es inválido.

Para que este sistema funcione, es imprescindible implementar una estrategia de doble token: un token de acceso (Access Token) de vida corta y uno de renovación (Refresh Token) de vida larga

```
bearer {
    loadTokens {
        BearerTokens(accessToken = "mi_token_jwt", refreshToken =
"")
    }
    refreshTokens {
        // Enviamos la petición de refresco como un mapa (JSON)
        //end point de refresco
        val response =
client.post("https://api.mi-app.com/auth/refresh") {
            markAsRefreshTokenRequest()
            //token de refresco
            setBody(mapOf("refresh_token" to "token...."))
        }

        if (response.status == HttpStatusCode.OK) {
            // Leemos la respuesta directamente como un Mapa
            val data = response.body<Map<String, String>>()

            // Extraer los valores usando las llaves del JSON
            val newAccess = data["access_token"] ?: ""
            val newRefresh = data["refresh_token"] ?: "antiguo
refresh token" ?: ""
            val idToken = data["id_token"] // Será null si es
OAuth, tendrá valor si es Google

            // actualizra el almacen de tokens
            // ...

            // Opcional: Si existe id_token, aquí se podriña
actualizar el perfilil
            if (idToken != null) {
                println("Se ha recibido identidad (OIDC):
```

```
$idToken")
            }

            BearerTokens(newAccess, newRefresh)
        } else {
            null
        }
    }
```

Dependiendo de los requerimientos del servidor, el refresh_token se puede enviar de dos formas principales:

Formato JSON (Estándar Moderno)

Es el más utilizado en APIs REST actuales. Se envía un objeto estructurado con el tipo de contenido application/json.

```
POST /auth/refresh HTTP/1.1
Host: api.mi-app.com
Content-Type: application/json

{
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5..."
}
```

Formato Estándar OAuth 2.0 (x-www-form-urlencoded)

Utilizado por servicios antiguos o proveedores de identidad estrictos. Los datos se envían como una cadena de parámetros.

```
POST /auth/refresh HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=eyJhbGciOiJIUzI1Ni...
```

Una vez procesada la renovación, el servidor responderá siguiendo uno de estos dos esquemas:

Oauth 2.0

Se limita a entregar las llaves de acceso a la API.

```
{
    "access_token": "at_987654321_oauth_only",
    "token_type": "Bearer",
    "expires_in": 3600,
    "refresh_token": "rt_optional_new_one"
}
```

OpenID Connect / Google (Autenticación)

```
{
    "access_token": "ya29.a0_google_access_token",
    "token_type": "Bearer",
    "expires_in": 3599,
    "refresh_token": "1//06_google_refresh_token",
    "id_token": "eyJhbGci... (JWT con perfil actualizado)"
}
```

Además de las llaves de acceso, incluye el id_token, que permite actualizar la identidad y el perfil del usuario localmente.

6.3. Almacenamiento del token

Hasta ahora, la gestión de tokens en el cliente se ha realizado de manera imperativa y manual. Este enfoque no solo incrementa la complejidad del desarrollo y la duplicidad de código, sino que constituye una vulnerabilidad crítica en términos de seguridad.

Una arquitectura robusta debe ser capaz de suministrar de forma transparente el token y su información asociada a los diferentes módulos de la aplicación. Para lograrlo, es fundamental implementar un sistema basado en dos pilares:

Desacoplamiento y Disponibilidad

La arquitectura debe garantizar que los tokens estén disponibles para el cliente HTTP de manera reactiva. El resto de la lógica de negocio no debería preocuparse por la inyección de cabeceras; esta tarea debe ser delegada a interceptores o plugins (como Ktor Auth) que soliciten la credencial solo cuando sea estrictamente necesario.

Custodia Segura de Credenciales

Al desarrollar aplicaciones con **KMP**, es necesario crear una capa de abstracción que permita gestionar estas credenciales de forma segura, independientemente de la plataforma en la que se ejecute el código. Para resolver este reto, la opción más sólida es integrar una librería que unifique el acceso a estos sistemas de almacenamiento seguro.

La librería elegida es **Multiplatform-Settings (https://github.com/russhwolf/multiplatform-settings)**, la cual permite almacenar datos en formato clave-valor de forma segura. Su principal ventaja es que utiliza automáticamente las tecnologías nativas de cada sistema:

Plataforma	**Tecnología Nativa Utilizada**	**Tipo de Almacenamiento**	**Seguridad**
Android	SharedPreferences / EncryptedSharedPreferences	XML en almacenamiento privado	Alta (con cifrado)

Plataforma	Tecnología Nativa Utilizada	Tipo de Almacenamiento	Seguridad
iOS / macOS	NSUserDefaults / Keychain	Property List / SQLite seguro	Muy Alta (Keychain)
Web (JS/Wasm)	localStorage	Claves/Valores del Navegador	Baja (No cifrado)
Windows	Registry / Credential Manager	Registro de Windows	Media
Linux	libsecret	Servicio de Secretos (Gnome Keyring)	Alta

Incluir la librería en la parte común de Gradle:

```
//almacenamiento seguro de token jwt
implementation("com.russhwolf:multiplatform-settings:1.3.0")
```

Incluir las librerías necesarias en cada plataforma, en el caso de Android/Desktop/Web solo es necesario en Android:

```
  androidMain.dependencies {
     ...
        //para guardar el token seguro
        implementation("androidx.security:security-crypto:1.1.0")

    }
```

Se define un almacén de claves utilizando la librería, en la capa de infraestructura:

```
class TokenStorage(private val settings: Settings) {
    companion object {
        private const val KEY_ACCESS_TOKEN = "access_token"
        private const val KEY_REFRESH_TOKEN = "refresh_token"
    }
    fun saveTokens(accessToken: String, refreshToken: String) {
        settings.putString(KEY_ACCESS_TOKEN, accessToken)
        settings.putString(KEY_REFRESH_TOKEN, refreshToken)
    }
    fun getAccessToken(): String? =
settings.getStringOrNull(KEY_ACCESS_TOKEN)

    fun getRefreshToken(): String? =
settings.getStringOrNull(KEY_REFRESH_TOKEN)

    fun clear() {
```

```
        settings.remove(KEY_ACCESS_TOKEN)
        settings.remove(KEY_REFRESH_TOKEN)
    }
}
```

Y se crea un singleton con Koin:

```
val InfraestructureModule = module {
    //almacenamiento del token
    single { TokenStorage(get()) }
```

TokenStorage recibe un objeto de tipo settings, que será concreto para cada plataforma, siendo necesario inyectar la configuración concreta en cada plataforma.

En Android se crea la clase AndroidPlataformModule se crea el settings para Android:

```
val AndroidPlatformModule = module {
    single<Settings> {
        val context = get<Context>()


        val masterKey = MasterKey.Builder(context)
            .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
            .build()


        val sharedPreferences = EncryptedSharedPreferences.create(
            context,
            "secret_shared_prefs",
            masterKey,

EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,

EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
        )


        SharedPreferencesSettings(sharedPreferences)
    }
}
```

Y al iniciar Koin para Android se carga el módulo:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        startKoin {
```

```
            modules(PresentationModule,
                DomainModule,
                InfraestructureModule,
                ApplicationModule,
                AndroidPlatformModule)
        }
        // FileKit.manualFileKitCoreInitialization(this)
        setContent {
            App()
        }
    }
}

@Preview
@Composable
fun AppAndroidPreview() {
    App()
}
```

Procediendo de forma similar con Jvm y web:

```
val DesktopPlatformModule = module {
    single<Settings> {
        val preferences = Preferences.userRoot().node("mi.app")
        PreferencesSettings(preferences)
    }
}
```

```
val WebPlatformModule = module {
    single<Settings> {
        StorageSettings() // Usa LocalStorage por defecto
    }
}
```

Un ejemplo de configuración final:

```
single {
    HttpClient {
        install(ContentNegotiation) { json() }

        install(Auth) {
            bearer {
                // Se recupera el almacén de tokens inyectado
                val tokenStorage = get<TokenStorage>()

                //Carga de credenciales desde el almacenamiento
```

```
seguro
                loadTokens {
                    val access = tokenStorage.getAccessToken()
                    val refresh = tokenStorage.getRefreshToken()

                    if (access ≠ null && refresh ≠ null) {
                        BearerTokens(access, refresh)
                    } else null
                }

                //Lógica de refresco automático ante errores 401
                refreshTokens {
                    // Se realiza la petición de renovación al
servidor
                    val response =
client.post("https://api.mi-app.com/auth/refresh") {
                        markAsRefreshTokenRequest()
                        setBody(mapOf("refresh_token" to
tokenStorage.getRefreshToken()))
                    }
                    if (response.status == HttpStatusCode.OK) {
                        val data = response.body<Map<String,
String>>()
                        val newAccess = data["access_token"] ?: ""
                        val newRefresh = data["refresh_token"] ?:
tokenStorage.getRefreshToken() ?: ""

                        // Actualización persistente en la
plataforma (EncryptedSharedPreferences/Keychain)
                        tokenStorage.saveTokens(newAccess,
newRefresh)

                        BearerTokens(newAccess, newRefresh)
                    } else {
                        // Si el refresco falla, se limpia el
almacén y se cierra la sesión
                        tokenStorage.clear()
                        null
                    }
                }

                // Filtrado de peticiones
                // Define qué rutas NO deben incluir el header de
autorización automáticamente
                sendWithoutRequest { request →
                    val path = request.url.pathSegments
```

```
                    // Se omiten rutas públicas o de autenticación
para evitar fugas de tokens
                    path.contains("public") ||
path.contains("auth")
                }
            }
        }
    }
}
```

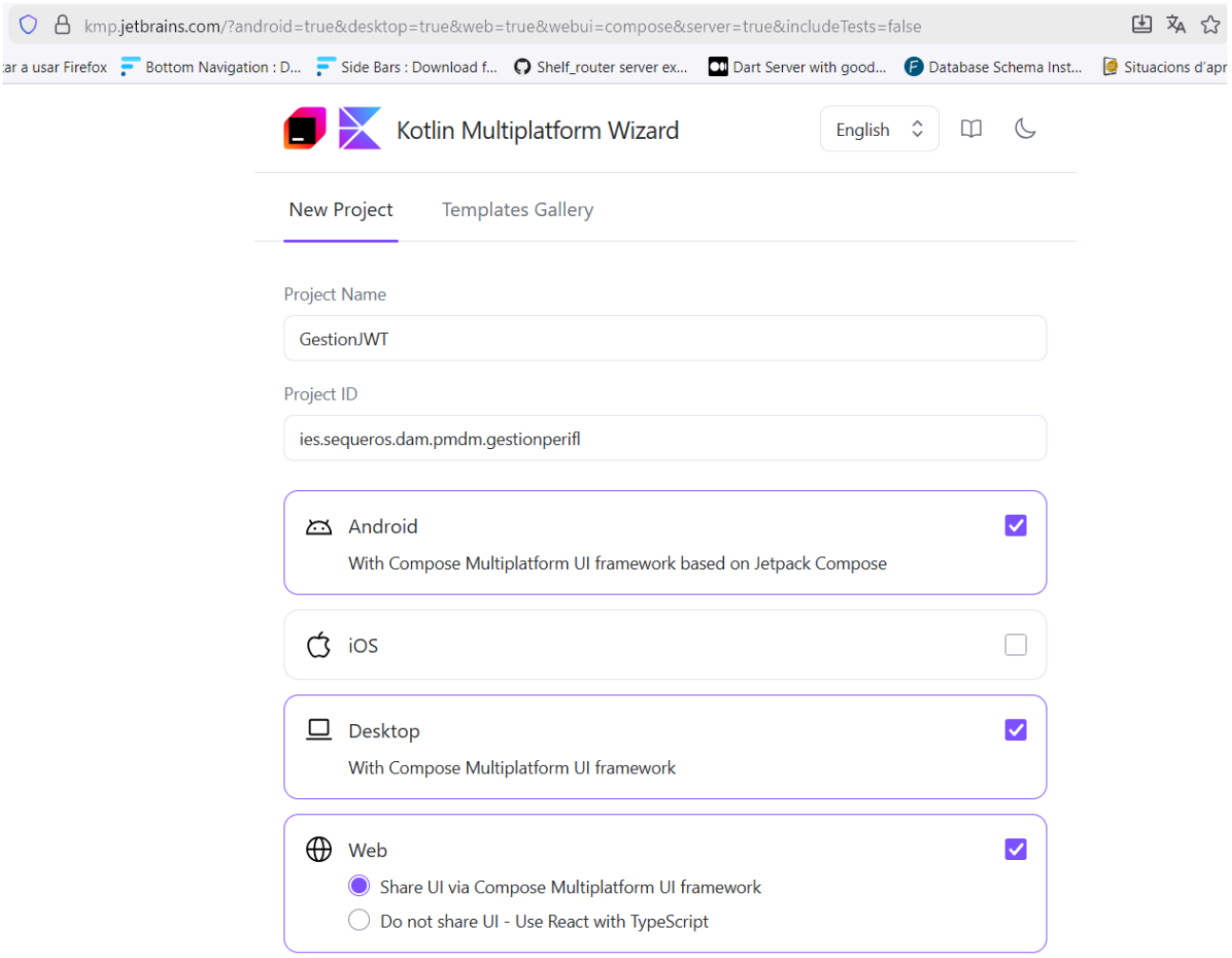
7. Ejemplo uso, gestión de usuarios

Prácticamente cualquier aplicación de uso generalizado gestiona los usuarios de forma similar, independientemente de lenguaje o:

1. Arranque: Buscar accessToken en el almacenamiento local.
2. Validación Local: Si no hay token, ir a Login. Si hay, comprobar fecha de expiración.
3. Validación Remota (Interceptores): Al hacer una petición, si el servidor devuelve 401 Unauthorized:
    La app intenta usar un refreshToken para pedir un nuevo accessToken.
    Si tiene éxito, repite la petición original (el usuario ni se entera).
    Si falla, borra todo y va a Login.

7.1. Configuración del entorno.

Crear el proyecto, con las 3 plataformas más el server:



ConfigurIncluir las librerías en el servidor para ktor:

libs.version.toml

```
[versions]
agp = "8.11.2"
android-compileSdk = "36"
android-minSdk = "24"
android-targetSdk = "36"
androidx-activity = "1.12.2"
androidx-appcompat = "1.7.1"
androidx-core = "1.17.0"
androidx-espresso = "3.7.0"
androidx-lifecycle = "2.9.6"
androidx-testExt = "1.3.0"
composeHotReload = "1.0.0"
composeMultiplatform = "1.10.0"
junit = "4.13.2"
kotlin = "2.3.0"
kotlinx-coroutines = "1.10.2"
ktor = "3.3.3"
logback = "1.5.24"
h2= "2.4.240"
```

```
postgres="42.7.8"
koin = "4.1.2-Beta1"


[libraries]
kotlin-test = { module = "org.jetbrains.kotlin:kotlin-test",
version.ref = "kotlin" }
kotlin-testJunit = { module =
"org.jetbrains.kotlin:kotlin-test-junit", version.ref = "kotlin" }
junit = { module = "junit:junit", version.ref = "junit" }
androidx-core-ktx = { module = "androidx.core:core-ktx",
version.ref = "androidx-core" }
androidx-testExt-junit = { module = "androidx.test.ext:junit",
version.ref = "androidx-testExt" }
androidx-espresso-core = { module =
"androidx.test.espresso:espresso-core", version.ref =
"androidx-espresso" }
androidx-appcompat = { module = "androidx.appcompat:appcompat",
version.ref = "androidx-appcompat" }
androidx-activity-compose = { module =
"androidx.activity:activity-compose", version.ref =
"androidx-activity" }
androidx-lifecycle-viewmodelCompose = { module =
"org.jetbrains.androidx.lifecycle:lifecycle-viewmodel-compose",
version.ref = "androidx-lifecycle" }
androidx-lifecycle-runtimeCompose = { module =
"org.jetbrains.androidx.lifecycle:lifecycle-runtime-compose",
version.ref = "androidx-lifecycle" }
kotlinx-coroutinesSwing = { module =
"org.jetbrains.kotlinx:kotlinx-coroutines-swing", version.ref =
"kotlinx-coroutines" }
logback = { module = "ch.qos.logback:logback-classic", version.ref
= "logback" }
ktor-serverCore = { module = "io.ktor:ktor-server-core-jvm",
version.ref = "ktor" }
ktor-serverNetty = { module = "io.ktor:ktor-server-netty-jvm",
version.ref = "ktor" }
ktor-serverTestHost = { module =
"io.ktor:ktor-server-test-host-jvm", version.ref = "ktor" }

ktor-client-content-negotiation = { module =
"io.ktor:ktor-client-content-negotiation", version.ref = "ktor" }
ktor-client-core = { module = "io.ktor:ktor-client-core",
version.ref = "ktor" }
```

```
ktor-client-okhttp = { module = "io.ktor:ktor-client-okhttp",
version.ref = "ktor" }
ktor-server-content-negotiation = { module =
"io.ktor:ktor-server-content-negotiation", version.ref = "ktor" }
ktor-serialization-kotlinx-json = { module =
"io.ktor:ktor-serialization-kotlinx-json", version.ref = "ktor" }
ktor-server-host-common = { module =
"io.ktor:ktor-server-host-common", version.ref = "ktor" }
ktor-server-resources = { module =
"io.ktor:ktor-server-resources", version.ref = "ktor" }
ktor-server-double-receive = { module =
"io.ktor:ktor-server-double-receive", version.ref = "ktor" }
ktor-server-auth = { module = "io.ktor:ktor-server-auth",
version.ref = "ktor" }
ktor-server-auth-jwt = { module = "io.ktor:ktor-server-auth-jwt",
version.ref = "ktor" }
ktor-server-statuspage = { module =
"io.ktor:ktor-server-status-pages", version.ref = "ktor" }
h2 = { module = "com.h2database:h2", version.ref = "h2" }
postgresql = { module = "org.postgresql:postgresql", version.ref =
"postgres" }
koin-ktor = { module = "io.insert-koin:koin-ktor", version.ref =
"koin" }
koin-logger-slf4j = { module = "io.insert-koin:koin-logger-slf4j",
version.ref = "koin" }

[plugins]
androidApplication = { id = "com.android.application", version.ref
= "agp" }
androidLibrary = { id = "com.android.library", version.ref = "agp"
}
composeHotReload = { id = "org.jetbrains.compose.hot-reload",
version.ref = "composeHotReload" }
composeMultiplatform = { id = "org.jetbrains.compose", version.ref
= "composeMultiplatform" }
composeCompiler = { id = "org.jetbrains.kotlin.plugin.compose",
version.ref = "kotlin" }
kotlinJvm = { id = "org.jetbrains.kotlin.jvm", version.ref =
"kotlin" }
ktor = { id = "io.ktor.plugin", version.ref = "ktor" }
kotlinMultiplatform = { id = "org.jetbrains.kotlin.multiplatform",
version.ref = "kotlin" }
kotlin-plugin-serialization = { id =
"org.jetbrains.kotlin.plugin.serialization", version.ref =
"kotlin" }
```

build.gradle.kts

```
plugins {
    alias(libs.plugins.kotlinJvm)
    alias(libs.plugins.ktor)
    application
}

group = "ies.sequeros.dam.pmdm.gestionperifl"
version = "1.0.0"
application {

mainClass.set("ies.sequeros.dam.pmdm.gestionperifl.ApplicationKt")

    val isDevelopment: Boolean = project.ext.has("development")
    applicationDefaultJvmArgs =
listOf("-Dio.ktor.development=$isDevelopment")
}

dependencies {
    implementation(projects.shared)
    implementation(libs.logback)
    implementation(libs.ktor.serverCore)
    implementation(libs.ktor.serverNetty)
    testImplementation(libs.ktor.serverTestHost)
    testImplementation(libs.kotlin.testJunit)
    implementation("io.ktor:ktor-server-cio:3.3.3")
    //inyección de dependencias
    implementation(libs.koin.ktor)
    implementation(libs.koin.logger.slf4j)

    //modulos de ktor
    implementation(libs.ktor.server.content.negotiation)
    implementation(libs.ktor.serialization.kotlinx.json)
    implementation(libs.ktor.server.statuspage)
    implementation(libs.ktor.server.resources)
    implementation(libs.ktor.server.double.receive)
    implementation(libs.ktor.server.auth)
    implementation(libs.ktor.server.auth.jwt)
    //jpa con hibernate
    implementation("org.hibernate.orm:hibernate-core:6.4.4.Final")

implementation("org.hibernate.orm:hibernate-hikaricp:6.4.4.Final")
    implementation(libs.postgresql)
    implementation(libs.h2)
```

```
implementation("jakarta.persistence:jakarta.persistence-api:3.1.0"
)
    //json-schema
    implementation("com.networknt:json-schema-validator:1.5.8")
    // codificar contrasenyas

implementation("org.springframework.security:spring-security-crypt
o:7.0.2")
    // necesita el login para codificar
    implementation("commons-logging:commons-logging:1.3.5")
    //cache de los repositorios
    implementation("com.github.ben-manes.caffeine:caffeine:3.1.8")
}
```

7.2. Definición y creación de la base de datos

Se crea una base de datos en PostGreSQL con una única tabla usando Docker, el fichero compose.yaml y el sql de creación se encuentra en la carpeta database.

7.3. Configuración inicial de Ktor

Se configuran los plugins de Content Negotiation (con serialización JSON) ,la inyección de dependencias con Koin, la posibilidad de leer varias veces el body de la petición y la posibilidad de disponer de contenido estático. Se opta por una configuración centralizada en el Application.module() para facilitar su lectura en este ejemplo, aunque en entornos productivos es preferible la modularización por características (features)

```
fun Application.module() {
    install(ContentNegotiation) {
        json()
    }
    install(Koin) {
        slf4jLogger()
        modules(appModulo)
    }
    install(DoubleReceive)
    routing {
        get("/") {
            call.respondText("Ktor: ${Greeting().greet()}")
        }
        staticFiles(
            remotePath = "/uploads",
            dir = File("uploads")
        )
    }
```

```
}
```

7.4. Definiendo los endpoints

La aplicación posee una API propia, básica que permite:

- Registrarse en el sistema
- Gestionar el perfil de usuario.
    - Modificar password.
    - Modificar imagen
    - Modificar nombre (en caso caso de no existir otro usuario con ese nombre)
    - Cambiar su estado, solo entre activo e inactivo.
    - Eliminar cuenta.
- Entra en el sistema con usuario y password.
- Renovar el token de acceso.

Los puntos con las peticiones y respuesta son, teniendo en cuenta que a partir de dominio la API se encuentra en /api son:

Endpoint	**JWT**	**Método**	**Request (Body JSON)**	**Response (Body JSON)**	**Comentario**
/public/register	No	POST	{"username", "email", "password"}	{"id", "username", "email"}	Registro inicial.
/public/login	No	POST	{"username", "password"}	{"access_token", "id_token", "expires_in", "token_type", "refresh_token"}	El id_token contiene los claims del perfil.
/public/refresh	No	PUT	{"refresh_token": "..."}	{"access_token", "id_token", "expires_in", "token_type", "refresh_token"}	El id_token devuelve el perfil actualizado.
/users/me	Sí	GET	Vacío	{"id", "username", "status", ...}	Datos extraídos de la DB tras validar el access_token.
/users/me/password	Sí	PUT	{"old_password", "new_password"}	{"message": "success"}	Cambio de credenciales.
/users/me/image	Sí	PUT/PATCH	Binary / Multipart	{"image_url": "..."}	Actualización de avatar.

/users/me	Sí	PATCH	{"username", "status"}	{"id", "username", "status", ...}	Modificación parcial de identidad.
/users/me	Sí	DELETE	{"password"}	Vacío (204)	Baja definitiva del usuario.

Se definen las rutas públicas y privadas, aunque de momento sin limitaciones.

```
routing {
    get("/") {
        call.respondText("Ktor: ${Greeting().greet()}")
    }
    route("api"){
        route("public"){
            post("register"){

            }
            post ("login"){
                call.respondText("login")
            }
            post("refresh"){
                call.respondText("refresh")
            }

        }
        route ("users"){
            route("me") {
                get("") {
                    call.respondText("get me")
                }
                put("password"){
                    call.respondText("change password")
                }
                put("image"){
                    call.respondText("change image")
                }
                patch(""){
                    call.respondText("change other attirbute")
                }
                delete(""){
                    call.respondText("delete")
                }
            }
        }
    }
```

7.5. Creando las capas de la arquitectura

La interfaz es solo un detalle de implementación. Una arquitectura robusta garantiza que la lógica central sea independiente de si el consumidor es una interfaz gráfica (GUI), un servicio web o una herramienta de consola, en los siguientes apartados se desarrollan cada una de las capas:

7.5.1. Capa de dominio

Contiene las entidades u objetos de dominio, la lógica del negocio y las interfaces de los repositorios, realizando una implementación sencilla sin ValueObject.

Clase User:

```
enum class UserStatus {
   PENDING, ACTIVE, INACTIVE, SUSPENDED
}
data class User(
   val id: UUID,
   var name: String,
   val email: String,
   var image: String? = null,
   var status: UserStatus = UserStatus.PENDING
) {
   fun isCanLogin(): Boolean = (this.status == UserStatus.ACTIVE
|| this.status== UserStatus.PENDING)
   fun activate() {
       this.status = UserStatus.ACTIVE
   }
   fun suspendAccount() {
       this.status = UserStatus.SUSPENDED
   }
   fun updateProfile(newName: String, newImage: String?) {
       if (newName.isNotBlank()) {
           this.name = newName
       }
       this.image = newImage
   }
}
```

La interfaz IUserRepository:

```
interface IUserRepository {
   //  Gestión del Perfil
   fun findById(id: UUID): User?
   fun findByEmail(email: String): User?
   fun findByUsername(username: String): User?
```

```
    fun create(user: User, password: String): User
    fun update(user: User): User
    fun delete(user: User): Unit
    fun exists(id: UUID): Boolean
    fun existsByName(name: String): Boolean
    fun existsByEmail(email: String): Boolean
    fun existsByNameOrEmail(name: String, email: String): Boolean
    // Gestión de Seguridad
    // Estas funciones manejan hashes de contraseñas sin que el
objeto User las vea
    fun updatePassword(userId: UUID, newPasswordHash: String):
Boolean
    fun updateImage(user:User):User
    fun getPasswordHash(userId: UUID): String?
    fun getPasswordHash(email:String): String?
}
```

La interfaz IFilesRepository para la gestión de los ficheros, en concreto el almacenamiento de la imagen de perfil.

```
class FilesRepository: IFilesRepository {
    override suspend fun save(
        entity: StorageEntity,
        fileName: String,
        id: String,
        bytes: ByteArray
    ): Pair<String, String> {
        // Obtener extensión
        val extension = File(fileName).extension
        if (extension.isBlank()) {
                throw kotlin.IllegalArgumentException("Extensión no
encontrada")
        }
        val nuevoNombre = "$id.$extension"
        val path=entity.path+nuevoNombre
        val file = File(path)
        file.parentFile.mkdirs()
        file.writeBytes(bytes)
        return Pair(nuevoNombre, path)
    }

    override suspend fun delete(entity: StorageEntity, fileName:
String) {
        val file= File(entity.path+fileName)
        file.delete()
```

```
    }

    override fun getImagePath(entity: StorageEntity, fileName:
String): String {
        if(fileName==null || fileName.isBlank()) {
            return ""
        }
        val path=entity.path+fileName
        return path
    }
}
```

Finalmente, se define la interfaz IPasswordEncoder dentro de la capa de dominio. Su implementación se delega a la capa de infraestructura para tratar el algoritmo de cifrado como un detalle de implementación, permitiendo que la capa de aplicación orqueste la seguridad de forma totalmente desacoplada.

```
interface IPasswordEncoder {
    fun encode(plainText: String): String
    fun matches(plainText: String, encodedPassword: String):
Boolean
}
```

7.5.2. Capa de infraestructura

La persistencia de datos se gestiona mediante Hibernate a través de la entidad UserJPA. La implementación del repositorio no solo extiende una clase base abstracta para centralizar la gestión de sesiones y eliminar código redundante, sino que también implementa el contrato de la interfaz definido en la capa de dominio. Esto garantiza que la lógica de negocio permanezca agnóstica a la tecnología de almacenamiento seleccionada.

Adicionalmente, en esta capa de infraestructura se definen:

- Excepciones específicas de persistencia y acceso a datos.

- Mapeadores (Mappers) para la transformación bidireccional entre objetos de dominio y entidades JPA.
- Implementaciones de almacenamiento para la gestión de ficheros (sistema de archivos).

UserJPA:

```
enum class UserStatus {
    pending,
    active,
    inactive,
```

```
    suspended
}
@Entity
@Table(name = "users")
open class UserJPA {
    @Id
    @Column(name = "id", nullable = false)
    open var id: UUID? = null

    @Column(name = "name", nullable = false)
    open var name: String? = null

    @Column(name = "email", nullable = false)
    open var email: String? = null

    @Column(name = "password", nullable = false)
    open var password: String? = null

    @Column(name = "image")
    open var image: String? = null

    @Enumerated(EnumType.STRING)
    @Column(name = "status", columnDefinition = "user_status")
    open var status: UserStatus = UserStatus.pending
}
```

Implementación de IPasswordEncoder:

```
class BCryptEncoderAdapter : IPasswordEncoder {
    private val encoder = BCryptPasswordEncoder()
    override fun encode(plainText: String): String {
        return encoder.encode(plainText).toString()
    }
      override fun matches(plainText: String, encodedPassword:
String): Boolean {
        return encoder.matches(plainText,encodedPassword);
    }
}
```

Repositorio:

```
    override fun create(user: User, password: String): User {
        var userJPA = UserMapper.toJPA(user, password)
        executeTransaction({ em →
            em!!.persist(userJPA)
```

```
        }, "crear")
        invalidateCache(this.tableName + "_all")
        return UserMapper.toDomain(userJPA)
    }

    override  fun update(user: User): User {
        var userJPA = UserMapper.toJPA(
            user
        )
        executeTransaction({ em ->
            em!!.createNamedQuery("User.updateUser").
            setParameter("status", user.status)
                .setParameter("name", user.name)
                .setParameter("id",user.id)
                .executeUpdate()
        }, "actualizar")
        invalidateCache(this.tableName + "_all")
        return UserMapper.toDomain(userJPA)
    }

    public override  fun updateImage(user: User): User {
        var userJPA = UserMapper.toJPA(
            user
        )
        executeTransaction({ em ->
            em!!.createNamedQuery("User.updateUserImage").
            setParameter("image", user.image)
                .setParameter("id",user.id)
                .executeUpdate()
        }, "actualizar imagen")
        invalidateCache(this.tableName + "_all")
        return UserMapper.toDomain(userJPA)
    }

     override  fun  updatePassword(userId:  UUID,  newPasswordHash:
String): Boolean {
        var item: UserJPA? = null
        executeTransaction({ em ->
            item = em?.find<UserJPA>(UserJPA::class.java, userId)
            if (item == null)
                                throw  EntityNotFoundException("User",
"getPasswordHash failed", Throwable("El ID no existe "))
            item?.password = newPasswordHash
            em!!.merge(item)
```

```
        }, "actualizar")
        return item ≠ null
    }

    override fun getPasswordHash(userId: UUID): String? {
        var item: UserJPA? = null
        executeTransaction({ em →
            item = em?.find<UserJPA>(UserJPA::class.java, userId)
              }, "obtener password")
        if (item == null)
              throw EntityNotFoundException("User", "getPasswordHash
failed", Throwable("El ID no existe "))

        return item!!.password?: ""
    }
    override fun getPasswordHash(email: String): String? {

        val cacheKey = this.tableName + email + "_email"
          val item: UserJPA? = executeQueryWithCache(cacheKey, { em
→
                              em!!.createNamedQuery("User.findByEmail",
UserJPA::class.java)
                .setParameter("email", email)
                .resultList
                .firstOrNull()
        }, "User.findByEmail")
        if (item == null)
              throw EntityNotFoundException("User", "getPasswordHash
failed", Throwable("El Email no existe "))
        return item.password?:null
    }

    override fun invalidateCache(key: User) {
        invalidateCache(this.tableName + "_all")
        invalidateCache(this.tableName + key.id + "_id")
        invalidateCache(this.tableName + key.email + "_email")
        invalidateCache(this.tableName + key.name + "_name")
              invalidateCache(this.tableName  +  key.id.toString()  +
"_exists_by_id")
             invalidateCache(this.tableName  +  key.name.toString()  +
"_exists_by_name")
            invalidateCache(this.tableName  +  key.email.toString()  +
"_exists_by_email")
         invalidateCache(this.tableName + key.email + "_" + key.name
+ "_exists_by_name_oremail")
```

```
    }
}
```

Mapper, con dos estilos: Java y Kotlin**:**

```
/*
Opcion 1 estilo java
*/
object  UserMapper {

     //  De  base  de  datos  a  Dominio  (Simplemente  ignoramos  el
password)
    fun toDomain(jpa: UserJPA): User {
        return User(
            id = jpa.id ?: UUID.randomUUID(),
            name = jpa.name ?: "",
            email = jpa.email ?: "",
            image = jpa.image,
            status = jpa.status
        )
    }

    // De Dominio a base de datos (Recibimos el hash externamente)
    fun toJPA(domain: User, encodedPassword: String?=null): UserJPA
{
        return UserJPA().apply {
            id = domain.id
            name = domain.name
            email = domain.email
                 password  =  encodedPassword  //  Aquí  inyectamos  el
password gestionado aparte
            image = domain.image
            status = domain.status
        }
    }

}
//opción 2, usando características de kotlin
fun UserJPA.toDomain(): User {
    return User(
        id = this.id ?: UUID.randomUUID(),
        name = this.name ?: "",
        email = this.email ?: "",
        image = this.image,
```

```
        status = this.status // Al ser el mismo Enum, la asignación
es directa
    )
}

fun User.toJPA(password:String?=null): UserJPA {
    val jpa = UserJPA()
    jpa.id = this.id
    jpa.name = this.name
    jpa.email = this.email
      // Si se pasa el password, se actualiza. Si es null, el
repositorio podría ignorarlo.
    password?.let { jpa.password = it }
    jpa.password = password
    jpa.image = this.image
    jpa.status = this.status
    return jpa
}
```

Repositorio de ficheros:

```
class FilesRepository: IFilesRepository {
    override suspend fun save(
        entity: StorageEntity,
        fileName: String,
        id: String,
        bytes: ByteArray
    ): Pair<String, String> {
        // Obtener extensión
        val extension = File(fileName).extension
        if (extension.isBlank()) {
                throw kotlin.IllegalArgumentException("Extensión no
encontrada")
        }
        val nuevoNombre = "$id.$extension"
        val path=entity.path+nuevoNombre
        val file = File(path)
        file.parentFile.mkdirs()
        file.writeBytes(bytes)
        return Pair(nuevoNombre, path)
    }

     override suspend fun delete(entity: StorageEntity, fileName:
String) {
        val file= File(entity.path+fileName)
```

```
        file.delete()
    }

    override fun getImagePath(entity: StorageEntity, fileName:
String): String {
        if(fileName==null || fileName.isBlank()) {
            return ""
        }
        val path=entity.path+fileName
        return path
    }
}
```

7.5.3. Aplicación

Esta capa contiene los Casos de Uso, que actúan como orquestadores de la lógica de aplicación. Reciben peticiones desde las capas externas (como los endpoints de la API), coordinan la interacción entre los repositorios, servicios y entidades de dominio, y devuelven DTOs (Data Transfer Objects). De este modo, se asegura que la capa de presentación reciba únicamente la información necesaria, manteniendo el dominio protegido y desacoplado de los contratos externos.
En esta capa se añaden las excepciones que pueden ocurrir como que el usuario ya exista en el caso de crear uno nuevo (usuario con mismo nombre o mail) o que se intente actualizar un usuario que no existe.

Los casos de uso que se definen son, junto con el comando que recibe y el dato devuelto:

Caso de Uso (Acción)	Comando / Query (Entrada)	DTO (Salida)	Comentario de Negocio
RegisterUser	RegisterUserCommand	UserDTO	Registro e inicialización de perfil.
LoginUser	LoginUserCommand	LoginDto	Autenticación y generación de tokens.
RefreshAuth	RefreshCommand	TokenDTO	Renovación de sesión con Refresh Token.
GetMyProfile	GetProfileQuery	UserDTO	Recupera perfil (ID extraído del JWT).
ChangePassword	ChangePasswordCommand	Unit (204 No Content)	Validación de pass antigua y nueva.

UpdateUserImage	UpdateImageCommand	UserDTO	Gestión de ficheros y URL de avatar.
UpdateProfile	UpdateProfileCommand	UserDTO	Actualización parcial (campos opcionales).
DeleteAccount	DeleteAccountCommand	Unit (204 No Content)	Baja del sistema tras validar pass.

Por ejemplo, el caso de uso de registro:

```
class RegisterUserUseCase(
   val repository: IUserRepository,
   val passwordEnconder: IPasswordEncoder,

   ) {
   suspend operator fun invoke(command: RegisterUserCommand):
UserDto =
       withContext(Dispatchers.IO) {
           //SE DELEGA LA COMPROBACIÓN EN QUE SALTE UNA EXCEPCIÓN
           //se crea el objeto del dominio a partir de comando
           val item = command.toDomain(UUID.randomUUID())
           val password =
passwordEnconder.encode(command.password)
               //se almacena en la base de datos
               val newUser = repository.create(item, password =
password)
               //se devuelve el dto
               UserDto.fromDomain(newUser)
       }
}
```

En el que se delega la comprobación de duplicados (nombre, mail) en la base de datos, que estos campos se han marcado como “unique”, saltando una excepción, **que es capturada por la capa de presentación y devolviendo el código http de conflictos (409).**

Recibe un comando:

```
@Serializable
data class RegisterUserCommand (val email:String,val username:
String, val password: String ){
```

```
}
fun RegisterUserCommand.toDomain(id: UUID):User{
    return User(id,username,email)
}
```

Y devuelve un UserDto, sin la **contraseña**:

```
@Serializable
data class UserDto(
    @Serializable(with = UUIDSerializer::class)
    val id: UUID,
    val username: String, val email: String, val image: String? =
null
) {
    companion object {
        fun fromDomain(user: User): UserDto {
            return UserDto(user.id, username = user.name, email =
user.email, user.image ?: null)

        }
    }
}
```

O el caso de uso de cambiar contraseña:

```
class ChangePasswordUseCase(
    val repository: IUserRepository,
    val passwordEnconder: IPasswordEncoder,

) {
    suspend operator fun invoke(id: UUID, command:
ChangePasswordCommand): Unit =
        withContext(Dispatchers.IO) {
            //obtener el password antiguo
            val oldPassword=repository.getPasswordHash(id)?:throw
InvalidCredentialsException("User")
            //comprobar si el que llega nuevo es diferente al que
ya existe

if(passwordEnconder.matches(command.newPassword,oldPassword)){
                    throw BusinessException("La nueva clave ha de ser
diferente a la anterior")
            }
            //comprobar que el que llega antiguo es igual al que
esta almacenado
```

```
            val
match=passwordEnconder.matches(command.oldPassword,oldPassword)
            if(!match){
                throw  InvalidCredentialsException("User")
            }
            //codificar
            val
newPassword=passwordEnconder.encode(command.newPassword)
            //guardar
            repository.updatePassword(id,newPassword)
        }
}
```

7.5.4. EndPoints

Capa de presentación, en este caso su funciones son:

- Validar las peticiones (request),
- Llamar a los casos de uso,
- Adaptar si es necesario los comandos a partir de la información que les llega en la petición,
- Enviar las respuestas en el formato correcto(response)
- Comprobar que se tiene permiso para ejecutar endpoint protegidos.

7.5.4.1. Configuración

Se configura Ktor con el plugin de seguridad con JWT:

Serialización:

```
fun Application.configureSerialization() {
    val install = install(ContentNegotiation) {
        json()
    }
}
```

Inyección de dependencias:

```
fun Application.configureKoin() {
    install(Koin) {
        slf4jLogger()
        modules(appModulo)
    }
}
```

Seguridad:

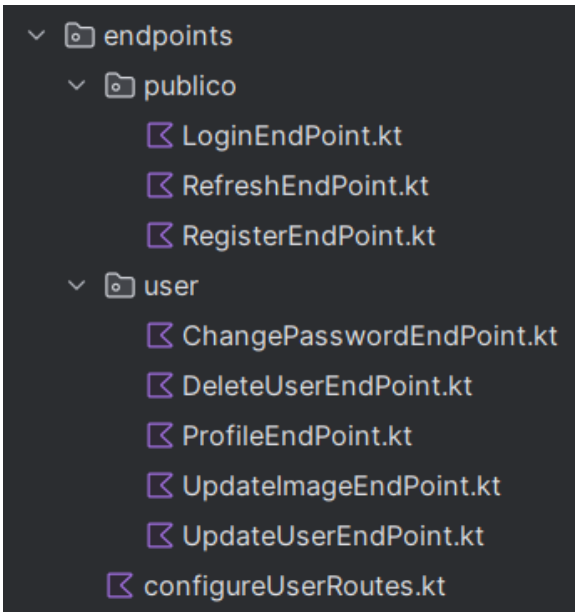
Definir los valores necesarios para JWT en el fichero de configuración (HOCON, YAML o Properties) en “resources”. Configurar el plugin de Ktor para usar Auth-JWT:

```
fun Application.configureSecurity() {
    // Extracción de parámetros para Usuario
    val userSecret =
environment.config.property("jwt.secret").getString()
    val userIssuer =
environment.config.property("jwt.issuer").getString()
    val userRealm =
environment.config.property("jwt.realm").getString()

    authentication {
        jwt("auth-user") {
            realm = userRealm
            verifier(
                JWT.require(Algorithm.HMAC256(userSecret))
                    .withIssuer(userIssuer)
                    .build()
            )
            //no se hacen comprobacione
            validate { credential ->
                //tiene que tener el claim subject
                if (credential.payload.subject != null) {
                    //se pasa la carga útil a los end points
                    JWTPrincipal(credential.payload)
                } else {
                    null //  autenticación falla
                }
            }
            //devuelve no autorizado
            challenge { defaultScheme, realm ->
                call.respond(HttpStatusCode.Unauthorized,"Token no
válido")          }
        }
    }
}
```

Rutas:

Se crean ficheros para cada una de las rutas, para poder configurar la validación json en cada endpoint.



La carga de las rutas, en las que se indica aquellas que necesitan autenticación (área):

```
fun Route.configureUserRoutes() {

    route("api") {
        route("public") {
            registerEndPoint()
            loginEndPoint()
            refreshEndPoint()
        }
        //para el acceso al endpoint se ha de tener un token válido
        authenticate("auth-user") {
            route("users") {
                route("me") {
                    changePasswordEndPoint()
                    deleteUserEndPoint()
                    profileEndPoint()
                    updateUserEndPoint()
                    updateUserImageEndPoint()
                }
            }
        }
    }
}
```

Y en la configuración general de rutas se llama a esta configuración, indicando además donde se encuentran los ficheros estáticos:

```
fun Application.configureRouting() {
    install(Resources)
    //permite leer varias veces el body, por ejemplo
```

```
    //validar el json del body y que pase al siguiente estado en
    //el pipeline
    install(DoubleReceive)
    //rutas
    routing {
        get("/") {
            call.respondText("Ktor: ${Greeting().greet()}")
        }
        //instanciar los endpoints del usuario
        configureUserRoutes()
        staticFiles(
            remotePath = "/uploads",
            //en caso de no existir la crea
            dir = File("uploads").apply { if (!exists()) mkdirs() }
        )
    }
}
```

Configuración global:

```
fun main(args: Array<String>): Unit = EngineMain.main(args)
fun Application.module() {
    //el orden importa
    //control de excepciones y http code status
    configureStatusPages()
    //Koin
    configureKoin()
    //configuración de json
    configureSerialization()
    //JWT
    configureSecurity()
    //routing
    configureRouting()
}
```

7.5.4.2. Desarrollo de los endpoints

Se deben implementar los diferentes endpoints del sistema. En aquellas rutas protegidas que requieran identificar al usuario, se extraerá la información necesaria (ID o email) directamente desde el **JWT (JSON Web Token)**. Siguiendo el principio de inversión de dependencias, los endpoints recibirán sus dependencias (como los casos de uso) mediante inyección, para lo cual es imprescindible la configuración previa de **Koin** (ver apartado 7.6).

Refresh:

```
fun Route.refreshEndPoint() {
    //se inyecta fuera para mejorar el rendimiento
    val refreshUserUseCase by inject<RefreshTokenUseCase>()
    route("refresh") {
        post() {
            val command = call.receive<RefreshTokenUserCommand>()
            var item = refreshUserUseCase(command)
            //en caso de no haber saltado ninguna excepción, se
devuelve el
            //objeto de refresco
            call.respond(HttpStatusCode.OK, item)

        }
    }
}
```

Delete:

```
fun Route.deleteUserEndPoint() {
    val deleteUseCase by inject <DeleteUseCase>()
    /**
     * para tener un plugin diferente
     */
    method(HttpMethod.Delete) {
        handle {
            //en el comando va la contrasenya
            val command = call.receive<DeleteCommand>()
            val principal = call.principal<JWTPrincipal>()
            //se obtiene el id necesario
            val id = principal?.subject ?: throw
InvalidCredentialsException("Error en credencial")
            val uuid = UUID.fromString(id)
            //se llama al caso de uso
            deleteUseCase(uuid, command)
            call.respond(HttpStatusCode.NoContent)
        }
    }
}
```

Observar cómo se obtiene el id del usuario a partir del JWT, en el comando se encuentra la contraseña antigua.

7.5.4.3. Control de excepciones y HTTP codes

En el desarrollo de APIs, la gestión de errores es una pieza crítica. Pueden surgir diversas excepciones durante el ciclo de vida de una petición: **usuarios duplicados, tokens expirados, credenciales inválidas o fallos de conexión con la base de datos entre otros.**

Para evitar que el servidor devuelva errores genéricos y poco informativos, se debe transformar estas excepciones en respuestas HTTP estandarizadas. Ktor facilita esta tarea mediante el plugin **StatusPages**, que permite interceptar excepciones en el *pipeline* de la aplicación, centralizar su tratamiento y definir códigos de estado específicos para el cliente.

Para instalarlo:

```
ktor-server-statuspage = { module =
"io.ktor:ktor-server-status-pages", version.ref = "ktor" }
```

```
implementation(libs.ktor.server.statuspage)
```

En un fichero se instala para que pase al **"Pipeline"** de Ktor, y se definen las excepciones a capturar y tratar:

```
fun Application.configureStatusPages() {
   //configuración
   install(StatusPages) {
       /**
        * EXCEPCIONES CAPA DE PRESENTACIÓN/ENDPOINTS
        */
       // Captura específicamente errores de campos faltantes
       exception<MissingFieldException> { call, cause →
           val missingFields = cause.missingFields.joinToString(",
")
           call.respond(
               HttpStatusCode.BadRequest,
               ErrorResponse(
                   error = "Datos incompletos",
                   detalles = listOf(
                       ValidationErrorDetail(
                           missingFields,
                            "Este campo es requerido por el
servidor."
                       )
                   )
               )
           )
       }
       exception<JsonConvertException> { call, cause →
           val internalCause = cause.cause
           val (mensajeAmigable, campoError) = when
(internalCause) {
               is MissingFieldException → {
```

```
                    "Faltan campos obligatorios en el JSON." to
internalCause.missingFields.joinToString(", ")
                }
                is SerializationException -> {
                    if
(internalCause.message?.contains("Encountered unknown key") ==
true) {
                        "El JSON contiene campos no permitidos por
el servidor." to "unknown_field"
                    } else {
                        "Error de formato o tipo de dato
incorrecto." to "payload"
                    }
                }
                else -> {
                    "El formato del JSON es inválido." to "generic"
                }
            }

            call.respond(
                HttpStatusCode.BadRequest,
                ErrorResponse(
                    error = "Error de serialización",
                    detalles = listOf(
                        ValidationErrorDetail(campoError,
mensajeAmigable)
                    )
                )
            )
        }
```

7.5.4.4. Validación de entrada

Para asegurar la integridad de los datos recibidos, Ktor permite gestionar la validación de dos formas: mediante el uso del plugin oficial Request Validation o a través de una implementación personalizada basada en esquemas JSON.

Validación Personalizada mediante JSON Schema

Esta opción es ideal cuando se busca desacoplar las reglas de validación del código Kotlin, utilizando archivos .json estándar almacenados en la carpeta resources.

El plugin personalizado (ubicado en ktor_config/plugins) intercepta la petición y se configura directamente en cada endpoint especificando la ruta del esquema:

```
install(ValidateSchema) {
```

```
            schemaPath =
"json_schemas/delete-user-command.schema.json"
        }
```

Plugin Oficial: RequestValidation

La alternativa nativa es el plugin RequestValidation. Este enfoque permite definir reglas de validación programáticas directamente en Kotlin, aprovechando el tipado fuerte de nuestras data classes.

Para utilizarlo, primero se ha de añadir la dependencia al proyecto:

```
ktor-server-validator = { module =
"io.ktor:ktor-server-request-validation", version.ref = "ktor" }
```

```
implementation(libs.ktor.server.validator)
```

Definiendo las validaciones para comando “request”:

```
fun Application.configureValidator() {
    install(RequestValidation) {
        val PASSWORD_REGEX =
Regex("^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$
!%*?&]{8,32}$")
        val EMAIL_REGEX =
Regex("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$")
        val JWT_REGEX =
Regex("^[A-Za-z0-9-_=]+\\.[A-Za-z0-9-_=]+\\.[A-Za-z0-9-_.+/=]*$")

        validate<UpdateUserCommand> { request →
            val validStatuses = UserStatus.entries.map {
it.name.lowercase() }
            when {
                request.name.length !in 2..100 →
                    ValidationResult.Invalid("El nombre debe tener
entre 2 y 100 caracteres")

                // Validación de Status (Debe estar en el Enum)
                //
!validStatuses.contains(request.status.lowercase()) →
                //    ValidationResult.Invalid("Estado no válido.
Valores permitidos: ${validStatuses.joinToString(", ")}")

                else → ValidationResult.Valid
```

```
            }
        }
        validate<RegisterUserCommand> { request →
            when {
                request.email.length !in 5..255 →
                    ValidationResult.Invalid("El email debe tener
entre 5 y 255 caracteres")
                !EMAIL_REGEX.matches(request.email) →
                    ValidationResult.Invalid("El formato del email
no es válido")
                request.username.length !in 8..64 →
                    ValidationResult.Invalid("El nombre de usuario
debe tener entre 8 y 64 caracteres")
                request.password.length !in 8..32 →
                    ValidationResult.Invalid("La contraseña debe
tener entre 8 y 32 caracteres")
                !PASSWORD_REGEX.matches(request.password) →
                    ValidationResult.Invalid("La contraseña debe
incluir mayúscula, minúscula, número y un carácter especial")

                else → ValidationResult.Valid
            }
        }
```

7.6. Inyección de dependencias

En el paquete di (Dependency Injection) se centraliza la configuración de los componentes que serán inyectados en el proyecto. Aquí se definen los módulos encargados de proveer instancias críticas, tales como el EntityManagerFactory para la persistencia, los repositorios de acceso a datos y el servicio de configuración global de la aplicación.

```
val appModulo = module {
    single<EntityManagerFactory> {
        try {

Persistence.createEntityManagerFactory("gestionJWTUnidadPersistenc
ia")
        } catch (e: Exception) {
            println("ERROR FATAL PERSISTENCIA: ${e.message}")
            throw DatabaseOperationException("Koin", "Error de
conexión: DB inaccesible", e)
        }
    }.onClose {
        //se cierra la factoria
```

```
        it?.close()
    }
    // Repositorios
    single<IUserRepository> {
        try {
            UserJPARepository(get())
        } catch (e: Exception) {
            println("ERROR FATAL PERSISTENCIA: ${e.message}")
            throw e
        }
    }
    //repositorio ficheros
    single<IFilesRepository> { FilesRepository() }

    //codificador
    single<IPasswordEncoder> { BCryptEncoderAdapter() }
    factory { RegisterUserUseCase(get(), get()) }
    factory { ChangePasswordUseCase(get(), get()) }
    factory { DeleteUseCase(get(), get(), get()) }
    factory { GetMyProfileUseCase(get(), get()) }
    factory { LoginUserUseCase(get(), get(), get()) }
    factory { RefreshTokenUseCase(get(), get()) }
    factory { UpdateUserCasoDeUso(get()) }
    factory { UpdateImageUserCasoDeUso(get(), get()) }
}
```

Y se instala en “plugin” de Koin el Ktor:

```
fun Application.configureKoin() {
    install(Koin) {
        slf4jLogger()
        val config = environment.config
        val jwtSecret = config.property("jwt.secret").getString()
        val jwtIssuer = config.property("jwt.issuer").getString()
        val jwtAudience =
config.property("jwt.audience").getString()
        //para poder inyectar la url en los endpoints en caso de
ser necesario
        //ruta de las imágenes
        val storageBaseUrl =
config.property("storage.baseUrl").getString()
        modules(module {
            single(named("baseUrl")) { storageBaseUrl }
            single<ITokenService> { TokenService(jwtSecret,
jwtIssuer, jwtAudience) }
            includes(appModulo)
        })
```

```
    }
}
```