
Proyecto de desarrollo de aplicaciones multiplataforma GESTION ENTRADAS DE CINE

CICLO FORMATIVO DE GRADO SUPERIOR
Desarrollo de Aplicaciones Multiplataforma (IFCS02)

Curso 2021-22

Autor/a/es:

JEREMY ALEXANDER RAMOS SEGURA

Tutor/a:

JOSE LUIS GONZALEZ SANCHEZ

Departamento de Informática y Comunicaciones
I.E.S. Luis Vives

1 CONTENIDO

2	INTRODUCCIÓN	3
	OBJETIVO	3
	ALCANCE	3
	Descripción del ptoyecto	4
3	ANALISIS	5
	REQUISITOS FUNCIONALES Y NO FUNCIONALES Y DE INFORMACION	5
	ANALISIS DEL MERCADO	7
	ANALISIS TECNOLÓGICO	8
	Ktor + Mongo	10
	Spring + Security	10
	Angular + Capacitor	11
	Flutter	12
4	Diseño	14
	Diseño del Diagrama	15
	Despliegue de aspectos variados	15
5	IMPLEMENTACIÓN	16
	Aspectos Varios	16
	Código de la aplicación:	17
	Config	17
	JWT	18
	WebSockets	18
	Controllers	20
	DTO'S	22
	ResultsError	22
	Exceptions	23
	Mappers	23
	Modelos	24
	Repositories	25
	Interfaz Flows Con Results	26
	Validators	27
	Properties Back	27
6	IMPLEMENTACIÓN FRONT-END	28
	Partes Para Destacar	29
	Router o ActivatedRoute	29
	Inyecciones en Angular	30
	Consumición del Back-End-API	30
7	RESULTADOS Y DISCUSIÓN	32
	Back-End	32
	Front-End	34
	Aspectos Varios	37
	Costes	37
8	BLIOGRAFÍA	37

2 INTRODUCCION

Como introducción al proyecto quiero recalcar que ya he tenido en cuenta la cantidad y gama de aplicaciones similares las cuales desempeñan el mismo papel. Pero como proyecto me pareció interesante ya que en estas aplicaciones hay algunas funcionalidades que para desempeñarlas como proyecto me resultan interesantes tales como la gestión de las reservas, la pasarela de pago etc. Por eso como opción al proyecto me resulto la más factible y teniendo en cuenta que el tema a tratar va de la mano con las películas será más ameno.



OBJETIVO

Con este proyecto pretendo tanto aprender nuevas tecnologías orientadas al desarrollo multiplataforma. A través de encontrarme con Frameworks y otras características las cuales no he visto. También desarrollarme en el mundo multiplataforma para tener más cerca el objetivo de un desempeño full-stack.



Reformar y adquirir nuevos conocimientos para que de forma autóctona me refuercen la base para entrar en el mundo laboral.

ALCANCE

- Establecer una comunicación simultanea en entre clientes y un servidor de forma íntegra.
- Realizar de forma correcta la gestión de validación mediante QR's.
- Envío en Tiempo Real de Actualizaciones de estados de las Butacas reservadas. (Mediante el mecanismo de webSockets)
- Exportar la aplicación al apartado web.

Servidor

Se encargará de gestionar las peticiones de uno o varios clientes a la vez y almacenará dichos datos dentro de una base de datos relacional.

Cliente

Web: Los cuales tendrán como funcionalidad el login de si mismos, la apreciación de un catálogo activo con las películas donde puedan realizar reservas a estas.

Descripción del ptoyecto

Como he indicado en mi anteproyecto tengo como objetivo proporcionar un sistema de gestión de entradas de cine. El programa contará con un **cliente** que se le mostrará al usuario el cual dispondrá de ciertas opciones **que puede ver estando logueado o no dentro del sistema**. Tales como: Ver en cartelera, Próximamente, junto con su respectiva portada, titulo, y tandas. **Sera una vista en cuadrícula para maximizar el atractivo de la aplicación**. Una vez seleccionada la película deseada aparecerá la opción de elegir hora de visita y a su dicha selección aparecerá el apartado de butacas donde el usuario seleccionará el numero de butacas que desea siempre y cuando no estén seleccionadas, una vez hecho aparecerá una pantalla de resumen para que el usuario vea sus opciones y se pase al método de pago. Una vez hecho el pago al usuario se le entregará las entradas con un **código QR** adjunto el cual se usará para validar la entrada. Una vez finalizada la selección de entradas, película y butacas la aplicación te indicará si quieres seguir en la compra por si le interesa comprar más entradas y en caso de estar con la sesión iniciada desloguear también nos llevaría al apartado de películas habiendo estado logueados o no dentro de la aplicación.

En el apartado del servidor todo los datos internos de la aplicación: (*Datos de películas, datos de usuarios entrantes en la app, registros de entradas y redistribución de las mismas*) se guardarán dentro del mismo en su respectiva base de datos. Por consiguiente, al cliente contar con un sistema de logueo el servidor gestionará esa entrada con un framework que expondremos más adelante. También la gestión de las entradas se tiene que gestionar en tiempo real para evitar disturbios y sobrecarga de selección de entradas. Cuyo comportamiento y/o funcionalidades serán consumidos por nuestro cliente.

3 ANALISIS

REQUISITOS FUNCIONALES Y NO FUNCIONALES Y DE INFORMACION

Requisitos Funcionales

Un requisito funcional es una declaración de cómo debe comportarse un sistema. Define lo que el sistema debe hacer para satisfacer las necesidades o expectativas del usuario. Los requisitos funcionales se pueden considerar como características que el usuario detecta. Son diferentes de los requisitos no funcionales, que definen cómo debe funcionar internamente el sistema (p. ej., rendimiento, seguridad, etc.).

Los requisitos funcionales se componen de dos partes: función y comportamiento. La función es lo que hace el sistema. El comportamiento es cómo lo hace el sistema.

- R-F1) Crear, modificar, eliminar y mostrar películas a elección.
- R-F2) Crear, modificar, eliminar y listar salas.
- R-F3) Crear, modificar, eliminar y listar usuarios.
- R-F4) Gestionar el sistema de generación de reservas.
- R-F5) Sistema de identificación de usuario.
- R-F6) Sistema de Registro de un Usuario.
- R-F7) Método de pago asociado al formato especificado: Tarjeta, Paypal etc.
- R-F8) Identificación del tipo de usuario para la gestión de ofertas.
- R-F9) Permitir búsquedas mediante filtros.
- R-F10) Validar la información introducida en el apartado de inicio de sesión.
- R-F11) El usuario tiene la capacidad de restablecer su contraseña en cualquier momento.
- R-F12) Permitir al usuario ver el resumen total de su pedido y/o modificar el mismo.
- R-F13) No permitir que el varios usuarios de forma simultanea tengan la misma butaca.
- R-F14) Generar una entrada impresa la cual mediante QR's se pueda validar.

Requisitos No Funcionales

Un requisito no funcional representa las restricciones o condiciones que impone el cliente al programa que necesita, por ejemplo, el tiempo de entrega del programa, el lenguaje o la cantidad de usuarios.

Los requisitos no funcionales de la aplicación son:

- RNF1) La aplicación cliente web y móvil ha de estar expuesta en Angular.
- RNF2) Cliente y servidor deben ser dos subsistemas distintos interconectados.
- RNF3) La parte del servidor tiene que albergar la posibilidad de gestionar varios clientes simultáneamente.
- RNF4) El lenguaje de programación en el servidor debe ser Kotlin.
- RNF6) El intercambio de información entre cliente y servidor será mediante WebSockets.
- RNF7) El servidor contará con una base de datos relacional para la gestión de reservas.
- RNF8) El servidor contará con una base de datos relacional para la gestión de Usuarios.
- RNF9) La aplicación contará con un inicio de sesión de usuarios.
- RNF10) La aplicación contará con un sistema de roles los cuales desempeñarán una acción.
- RNF11) Las entradas se validarán mediante un sistema QR.
- RNF12) La aplicación contará con un inicio de sesión de usuarios Mediante tokens.
- RNF13) No es necesario que el usuario esté logueado para ver las butacas.
- RNF14) El usuario tiene que estar logueado para comprarlas.
- RNF15) La Base de datos de la aplicación estará regulada mediante SQL.

Requisitos De Información

RI1) Usuario. Es aquel que ejecuta la acción de interactuar con nuestra aplicación. Cada usuario tendrá un rol en este caso admin o usuario normal. El admin realizará acciones de actualización del listado de películas y gestión de los usuarios dentro de la aplicación, así como la gestión de entradas salas, etc. Y un usuario puede hacer reservas y consultar catálogo. Tiene como parámetros: Nombre usuario, email, contraseña, tarjeta , monto, rol, estado.

RI2) Películas. Son las distintas variantes que se mostrarán en un catálogo. Tienen como parámetros: Nombre, Portada, Directores, Actores, Duración, Categorías, Descripción y estado.

RI3) Salas. Son los diferentes espacios los cuales albergarán los usuarios para el disfrute de la película. Tiene como parámetros: Nombre, Tipo, Numero de Butacas, Tipo de butacas. Fila, Butaca,

RI5) Reserva. El resumen total con los parámetros de gestión. Tiene como parámetros: Nombre, Nombre Usuario, Identificador, Fecha, estado, Película, Sala, Numero Butacas, Total.

RI6) Entradas. Modelo el cual presentará la entrada una vez realizada la reserva. Tiene como parámetros: Nombre Película, Sala, Fecha, Hora, Fila, Butaca, Tipo Sala, Precio, Código QR.

RI7) Butacas. Son los componentes de las salas que tendremos en el cine. Tienen como parámetros: Fila, Numero, Tipo.

ANALISIS DEL MERCADO

Como ya sabemos tenemos como base que partimos de que ya existen aplicaciones similares una de estas es la aplicación del **cinesa**.



Características de la app

Dicha aplicación cuenta con un sistema similar a lo expuesto en este proyecto ya que cuenta con un apartado del login el cual se complementa de usuario y contraseña con la posibilidad de registrarse y/o cambiar contraseña. También cuentan con la implementación de la gestión de las butacas por lo visto en tiempo real lo que conlleva un punto muy fuerte de la aplicación. También cuenta con un buscador de cines lo cual lo hace independiente ya que podremos localizar cualquier cine de la franquicia y hacer lo pertinente sobre la aplicación.

INICIAR SESIÓN

Correo electrónico

Contraseña

☐ Recuérdame

☐ No soy un robot

Entrar

[Recordar contraseña](#)

[¿No tienes una cuenta? Regístrate aquí](#)

[Crear una cuenta](#)

Buscar

Mostrando 37 cines (puedes seleccionar hasta 5)

- As Cancelas**
C.C. As Cancelas, Local 2.01, Av. Do Camiño Frances
- Bahía de Santander**
C/ Francisco, Tomás y Valiente 1, 39011 Santander
- Barnasud**
Carrer Progrés 69, 08850 Gavá, Barcelona
- Bonaire**
Parque Comercial , Bonaire, -Ctra. NIII Km. 345
- Camas**
C.C. Carrefour Aljarafe, C/ Poeta Muñoz San Román S/N
- Diagonal**
Carrer de Santa Fe, de Nou Mèxic s/n, 08017 Barcelona
- Diagonal Mar**
CC Diagonal Mar, planta 3, Avda. Diagonal, 3, 08019 Barcelona



ANALISIS TECNOLÓGICO

Aquí se expondrán los distintos puntos de vista desde el nivel tecnológico de los cuales he elegido las tecnologías pertinentes que voy a utilizar en mi aplicación.

Base de Datos

Bases de Datos relacionales

Las bases de datos relacionales se caracterizan por ser una colección ordenada de registros que se organizan en un conjunto de tablas. Estas tablas se relacionan entre sí.

Para acceder a estos datos, usaremos lo que se conoce como Lenguaje de Consultas Estructuradas, (SQL, Structured Query Language). Con SQL podemos obtener y alterar datos de una forma organizada siempre y cuando tengamos en cuenta cuál es la estructura de la base de datos con la que estamos trabajando.

Las bases de datos relacionales se organizan a través de identificadores. De este modo, cada tabla tiene un identificador único que es el que va a establecer su relación con el resto de tablas. A su vez, estos identificadores hacen que sea más fácil organizar cada una de las tablas por separado.

Los principales sistemas gestores de bases de datos relacionales son: MySQL, MariaDB, SQLite, PostgreSQL, SQL Server y Oracle.



Bases de datos no relacionales

Las bases de datos no relacionales están diseñadas para modelos de datos específicos y que no necesitan ser relacionados con otros modelos. Cada tabla funciona de forma independiente y son mucho más sencillas que los modelos relacionales.

Las bases de datos no relacionales pueden tener identificador único, es decir, para identificar cada uno de los registros de la base de datos, pero este identificador no se usará (generalmente) para relacionar unos registros con otros. Como veremos, la información se organiza normalmente mediante documentos y es muy útil cuando no tenemos un esquema exacto de lo que se va a almacenar.

Con respecto a los formatos que se utilizan en las bases de datos no relacionales, podríamos decir que el formato más popular es el del documento. En muchos casos, lo que se utiliza es un objeto con una clave y un valor para que el acceso a la información sea pueda realizar de una forma sencilla.

Los principales sistemas gestores de bases de datos no relacionales son: MongoDB, Redis y Cassandra.



Elección

En este caso me decanté por el sistema de base de datos relacional ya que en función de lo que tengo planteado para mi aplicación tanto gestión de reservas, como de los distintos modelos representativos de la información, será mas provechosa que una base de datos Relacional. También en la gestión de usuarios será provechoso.



Back-End

En este caso tengo Dos opciones: **Ktor y SpringBoot**

Mi elección en este caso ha sido Spring + Spring Security, ya que por consiguiente a lo expuesto anteriormente en el apartado de introducción las tecnologías expuestas en este programa me gustarían sentar una base para que en el entorno profesional prospere como programador y pienso que spring será una gran oportunidad.

Ktor + Mongo



Podríamos utilizar 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.

Como opción a la base de datos tenemos MongoDB 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 NoSQ.

Spring + Security

Podríamos utilizar Spring mas especifico SpringBoot para el manejo Api's. 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 UserDetailsServiceImpl. 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.



Front-End

Estudiando las alternativas multiplataforma, y que tenga algún interés en aprender de este apartado he dado con la conclusión que tengo dos opciones Angular + Capacitor o Flutter.

Angular + Capacitor

Angular es un framework Javascript potente, muy adecuado para el desarrollo de aplicaciones frontend modernas, de complejidad media o elevada. El tipo de aplicación Javascript que se desarrolla con Angular es del estilo SPA (Single Page Application) o también las denominadas PWA (Progressive Web App).

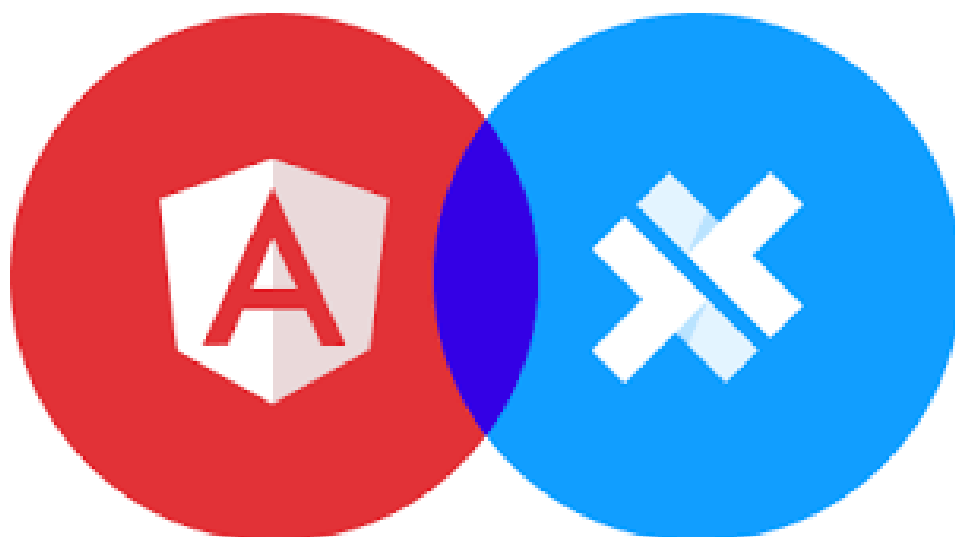
El framework Angular ofrece una base para el desarrollo de aplicaciones robustas, escalables y optimizadas, que promueve además las mejores prácticas y un estilo de codificación homogéneo y de gran modularidad.

El desarrollo en Angular se hace por medio de TypeScript (aunque también se podría desarrollar con Javascript, todas las guías y recomendaciones se basan en usar TypeScript), un superset del lenguaje Javascript que ofrece muchas herramientas adicionales al lenguaje, como el tipado estático o los decoradores.

Capacitor

Ionic es un framework para el desarrollo de aplicaciones híbridas. Con Ionic podemos construir aplicaciones para móviles Android, iOS, así como PWA y aplicaciones de escritorio, con una única base de código, aprovechando tus conocimientos de HTML, CSS y Javascript.

Gracias a Capacitor es muy sencillo acceder al SDK nativo de cada plataforma, con una interfaz de desarrollo unificada y optimizada para su uso con el framework Ionic. Capacitor es potente, sencillo y extensible vía plugins creados por el propio equipo de Ionic y la enorme comunidad de este framework.



Flutter

Flutter es el kit de herramientas de UI de Google para realizar hermosas aplicaciones, compiladas nativamente, para móvil, web y escritorio desde una única base de código.

Algunas características son:

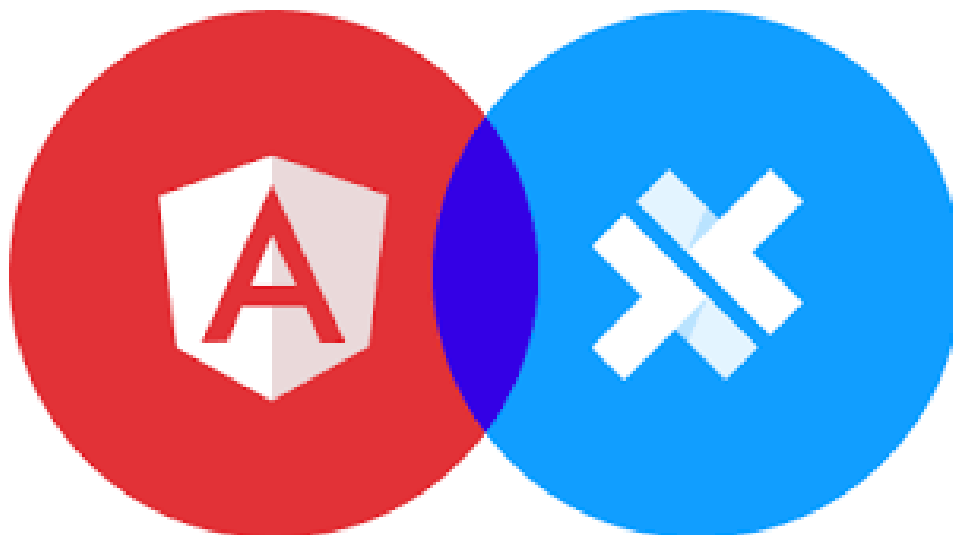
-**Desarrollo ágil**, con features como Hot reload, que te permite hacer cambios y ver las actualizaciones de manera instantánea.

-**Apps nativas con una misma base de código**. A diferencia de otros frameworks donde la gran parte del código se comparte, en Flutter el 100% del código funciona para ambas plataformas, y no tienes que escribir código personalizado para ninguna, esto mientras compila a aplicaciones nativas.



Elección

A mi criterio esta elección es muy difícil ya que si comparamos los pros y contras de cada framework flutter cuenta con unas ventajas sobre angular. Pero me puse a pensar y viendo que en el entorno profesional el cual me desenvolveré por ahora la preferencia de las aplicaciones es en angular y simplemente por el motivo de la opción a aprender tecnologías referentes a lo esencial actualmente me decanto por angular. También la idea del capacitor me llamo mucho la atención con lo cual sí, me decanto por angular. También de forma simultanea en las prácticas estaba desempeñando la creación de interfaces mediante angular con lo cual compaginar estas tecnologías será ameno.

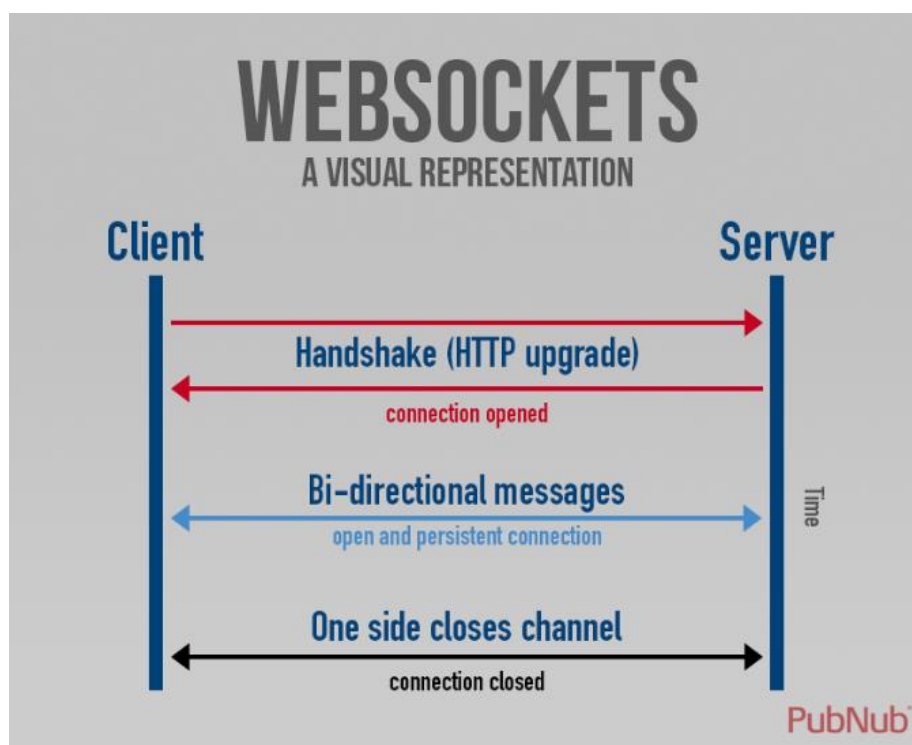


WebSockets

Como funciona WebSocket:

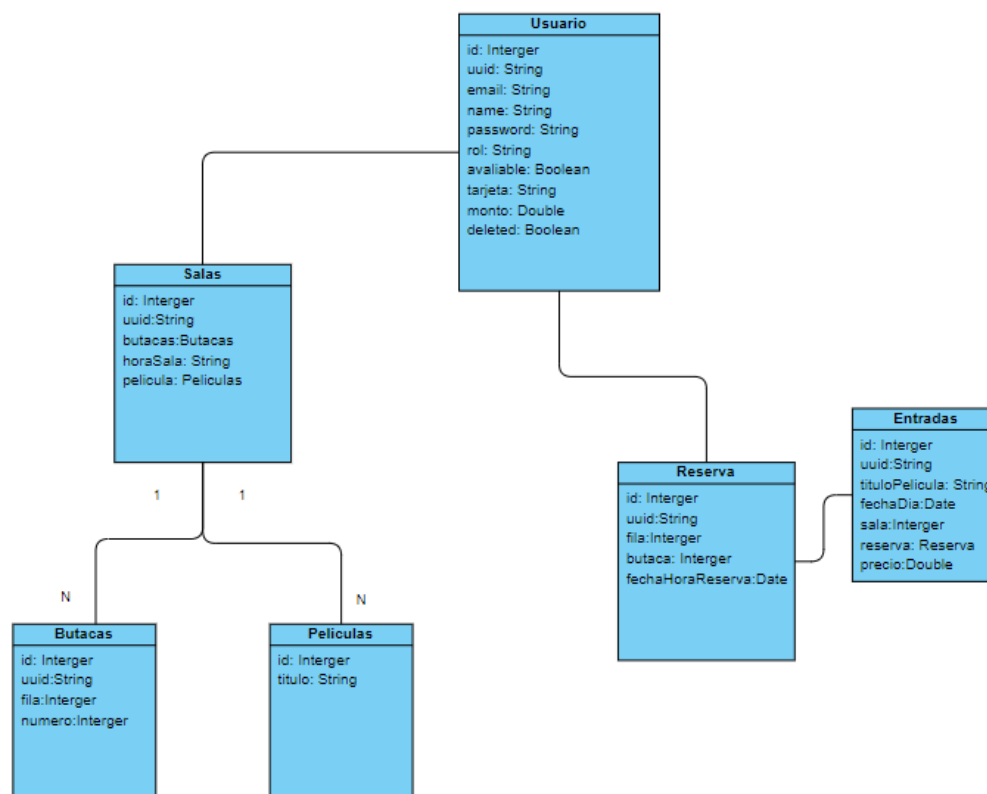
La comunicación a través de plataformas web está basada en HTTP, la cual permite requerir información desde un cliente, usualmente un navegador, a un servidor, por ejemplo en Node.js. El problema surge cuando queremos hacer el proceso inverso. WebSocket resuelve el problema permitiendo que la información se transmita a través de mensajes en los dos sentidos en una misma conexión.

Esto nos trae muchas ventajas, como la reducción de carga a través de la red y principalmente una arquitectura que nos permite diseñar nuestros proyectos de una manera más sencilla, minimizando los problemas que nos podemos encontrar. Ahora bien, suele ser muy conveniente utilizar abstracciones que trabajan encima de WebSocket al desarrollar nuestras aplicaciones, ya que ellas nos proveen de utilidades ya pensadas para resolver los problemas más comunes a ocurrir.



4 DISEÑO

Diagrama De Clases



Explicación del Diagrama

En el diagrama apreciamos que nuestro sistema se compone de usuarios los cuales pueden visualizar de forma abierta las salas (1-1) donde dentro de cada sala dentro de esta se pueden reproducir en un día varias películas las cuales cuentan con su título el cual solamente se añadirá al realizar la reserva.

Dentro de la sala también nos encontramos con butacas, las cuales cuentan con su identificador y su número butaca junto con la fila en la que se encuentra.

De otra manera un usuario puede realizar una reserva la cual está asociada a las butacas donde seleccionarán una butaca y dependiendo de las directrices de la reserva se le brindará la entrada o no.

También en el caso de las entradas tenemos que cada entrada tendrá su identificador y tendrá el título de la película, junto a que conlleva en sí su respectiva hora de creación, la reserva, el precio y un número el cual será el de la sala y que en físico una vez impresa en su parte posterior contará con un QR para validar la misma.

Diseño del Diagrama

Este apartado trata sobre la creación de los diagramas y bocetos varios los cuales se estarán propiciados en esta documentación.



Despliegue de aspectos variados

Docker

Docker es una plataforma de contenedores que permite empaquetar y distribuir entornos aislados y portátiles. Los contenedores son unidades de software livianas y autónomas que encapsulan todo lo que hace falta para ejecutar una aplicación, incluidas bibliotecas, herramientas y archivos de configuración, proporciona una forma eficiente y portátil de desplegar bases de datos, ofreciendo ventajas como la portabilidad, el aislamiento, la escalabilidad y la gestión de versiones.

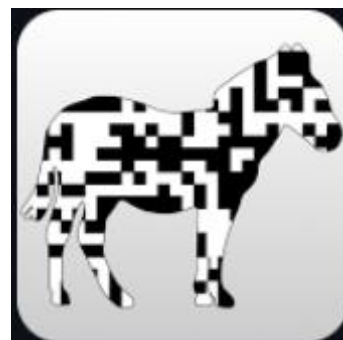


5 IMPLEMENTACIÓN

Aspectos Varios

QR

Un aspecto a tener en cuenta que he dicho que tendré que implementar será el tema de los QR's ya que le dará realismo a la aplicación y a su contenido de salida. Para ello he investigado y he encontrado una solución la cual consta de introducir una librería externa sobre mi proyecto llamada Zxing.



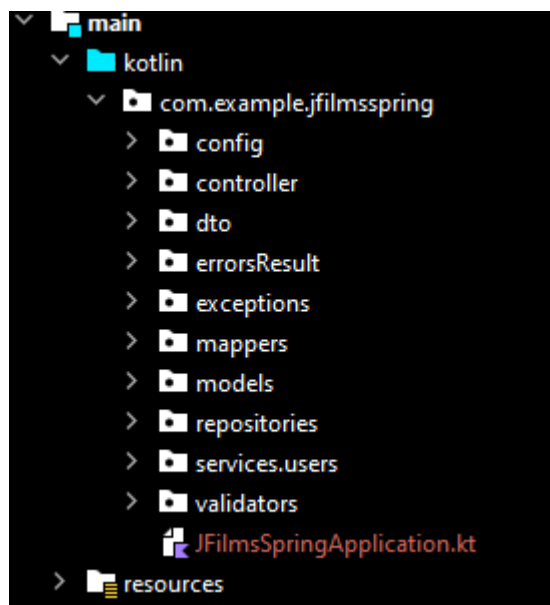
Plataforma de Pago

En este tema he tenido que improvisar ya que para hacer que la aplicación contenga metodología de pago totalmente real tendría que pagar una pasarela de plataforma de pago para asociarla con mi proyecto y así poder representar el método de pago a través de esta. Lo que he decidido es: Y si simulo la pasarela?. Que fue idea de mi tutor y me decidí por ello con lo cual por esa razón nuestro usuario en sus datos cuenta con una tarjeta y un monto el cual aumentará o disminuirá dependiendo de las reservas que se hagan o sobre las butacas. Lo haré todo mediante base de datos ya que es simulado y da la apariencia de realismo.



Código de la aplicación:

La aplicación se compone de una arquitectura clave para su entendimiento y que sea lo más legible posible hacia las personas que la mantenga. La estructura es esta:



Donde cada paquete hará referencia a las clases que tenemos ya expuestas dentro del mismo. La aplicación trabaja con la gestión de apis por eso vemos controladores, validadores, results etc.

Aquí la explicación de cada paquete:

Config

En este apartado expongo las configuraciones previas de mi programa.

Como primera opción tenemos el apartado de **APIConfig** el cual se caracteriza por tener la url de la api en caso de cambio.

```
@Configuration
class APIConfig {
    companion object {
        @Value("api")
        const val API_PATH = "/api"
    }
}
```

SecurityConfig

En este apartado se exponen los accesos a la api junto con los Authentication Manager el cual configurará el acceso a nuestra api también utilizaremos el JWT para la creación y el manejo de tokens para acceder a las peticiones pertinentes.

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

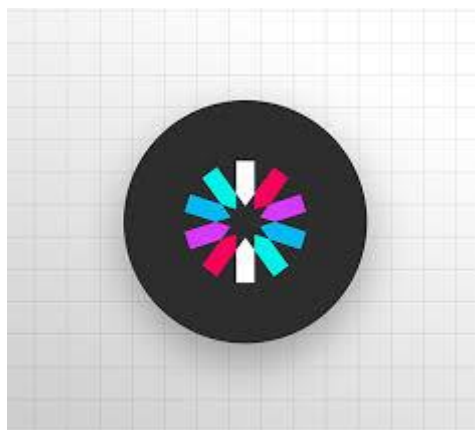
    http
        .exceptionHandling() { ExceptionHandlingConfigurer<HttpSecurity!>!! }
        .and() { HttpSecurity! }
        .authenticationManager(authenticationManager)
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) { SessionManagementConfigurer<HttpSecurity!>!! }
        .and() { HttpSecurity! }
        .authorizeHttpRequests() { AuthorizeHttpRequestsConfigurer<HttpSecurity!>.AuthorizationManagerRequestMatcherRegistry! }
        .requestMatchers(...patterns: "/error/**").permitAll()
        // .requestMatchers("/api/**").permitAll()
        .requestMatchers(...patterns: "/api/users/login", "/api/users/register", "/api/peliculas/todas", "/api/butacas").permitAll()
        .and() { HttpSecurity! }
        .addFilter(JwtAuthenticationFilter(jwtTokenUtil, authenticationManager))
        .addFilter(JwtAuthorizationFilter(jwtTokenUtil, userService, authenticationManager))
        .csrf() { CsrfConfigurer<HttpSecurity!>!! }
        .disable()

    return http.build()
}
```

JWT

Un JWT consta de tres partes separadas por puntos: el encabezado (header), la carga útil (payload) y la firma (signature). Cada una de estas partes se codifica en Base64URL y se concatenan con puntos para formar el JWT completo.

Cuando se envía un JWT a un servidor, el servidor puede verificar la integridad del token extrayendo el encabezado y la carga útil, generando la firma utilizando la clave secreta correspondiente y comparándola con la firma incluida en el token. Si las firmas coinciden, se puede confiar en que el token no ha sido modificado y se puede acceder a la información contenida en él.



WebSockets

Como por siguiente me pongo a relatar el tema de los webSockets en este caso tenemos una clase **ServerWebSocketConfig** la cual aplicamos el websocket sobre los siguientes endpoints: **Butacas, Reservas, Salas.**

Dentro de sí tenemos los handlers los cuales nos harán de intermediario entre los clientes y el websocket.

Con esto vamos a controlar las sesiones conectadas para poder establecer la conexión en tiempo real con la aplicación.

Por consiguiente, explico la creación de los handler:

En este caso tenemos la creación de las sesiones por cada conexión las cuales almacenaremos en una lista para que cada sesión se encuentre dentro del propio servidor dentro de la lista.

```
private val sessions: MutableSet<WebSocketSession> = CopyOnWriteArraySet()
```

Con cada conexión nos aparecerá ciertos mensajes y se añade la sesión a la lista.

```
override fun afterConnectionEstablished(session: WebSocketSession) {  
    logger.info { "Bienvenido a JFilms " }  
    logger.info { "Sesión: $session" }  
    sessions.add(session)  
    val message = TextMessage(payload: "Que se le ofrece")  
    session.sendMessage(message)  
}
```

También tenemos un método para enviar el mensaje establecido a cada sesión.

```
override fun sendMessage(message: String) {  
    logger.info { "Enviar mensaje de cambios en $entity: $message" }  
    sessions.forEach { session ->  
        if (session.isOpen) {  
            session.sendMessage(TextMessage(message))  
        }  
    }  
}
```

Controllers

Aquí tenemos la parte crucial de nuestro back que en este caso es las llamadas a la api desde los controladores pertinentes.

Como ejemplo comenzamos a explicar el controlador de **usuarios**.

Tenemos varios apartados los cuales son importantes como primero tenemos un login y un register los cuales van referidos a los usuarios y Admins de la aplicación.

```
@PostMapping("/login")
suspend fun loginUser(@Valid @RequestBody loginDto: UsuarioLoginDto): ResponseEntity<UsuarioWithTokenDto> {
    logger.info { "logueando" }
    val user = UsuarioService.loadUserByEmail(loginDto.email)
    val jwtToken: String = jwtTokenUtil.generateToken(user)
    logger.info { "Token: ${jwtToken}" }
    val userToken = UsuarioWithTokenDto(user.toDto(), jwtToken)
    return ResponseEntity.ok(userToken)
}
```

Como versión especial tenemos un método para ver nuestro perfil en caso de solamente verlo

```
@PreAuthorize("hasAnyRole('USER','ADMIN')")
@GetMapping("/me")
fun meInfo(@AuthenticationPrincipal user: Usuario): ResponseEntity<UsuarioDto> {
    logger.info { "Obteniendo la información del usuario : ${user.name}" }
    return ResponseEntity.ok(user.toDto())
}
```

y/o actualizar mi propio usuario. Las llamadas pertinentes tienen como etiqueta el preAuthorized para que solamente los usuarios con Token puedan acceder a estas llamadas para control de seguridad.

```
@PreAuthorize("hasAnyRole('USER','ADMIN')")
@PutMapping("/me")
suspend fun updateMe(@AuthenticationPrincipal user: Usuario, @Valid @RequestBody UsuarioDto: UsuarioUpdateDto):
    ResponseEntity<UsuarioDto> {
    logger.info { "Actualizando usuario: ${user.name}" }
    UsuarioDto.validate()
    val userUpdated = user.copy(
        email = UsuarioDto.email,
        name = UsuarioDto.name,
        dni = UsuarioDto.dni
    )
    try {
        val userUpdated = UsuarioService.update(userUpdated)
        return ResponseEntity.ok(userUpdated.toDto())
    } catch (e: Exception) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    }
}
```

Como segundo ejemplo tenemos ButacasController que este controlador se encargará de controlar la creación y asignación de butacas a una sala.

```
@PostMapping("/crear")
suspend fun create(@Valid @RequestBody butacaDto: ButacaCreateDto): ResponseEntity<ButacaDto> {
    butacaDto.validate().andThen { it: ButacaCreateDto
        println("pasa")
        butacasService.save(it.toModel())
    }.mapBoth(
        success = { it: Butacas
            return ResponseEntity.status(HttpStatus.CREATED)
                .body(it.toDto(salasService.findById(it.salaId).get()!!))
        },
        failure = { return handleErrors(it) }
    )
}
```

En este caso vemos que de forma correcta devolvemos la respuesta pertinente y en otro caso devolvemos un `handleError`. En este caso estamos haciendo uso del apartado de los Results para controlar lo pertinente de entrada y salida sobre los endpoints junto a ese método tenemos un conjunto de atributos los cuales son los errors sobre cada objeto. Donde relatamos que según el error devolveremos un `NOT_FOUND` o un `BAD_REQUEST` en caso de fallo.

```
private fun handleErrors(butacasError: ButacasError): ResponseEntity<ButacaDto> {
    when (butacasError) {
        is ButacasError.NotFound -> throw ResponseStatusException(
            HttpStatus.NOT_FOUND,
            butacasError.message
        )

        is ButacasError.BadRequest -> throw ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            butacasError.message
        )
    }
}
```

DTO'S

Siguiendo con las butacas ya que básicamente los controladores son iguales simplemente controlamos la entrada y salida de cada uno. Siguiendo con el ejemplo anterior sobre cada método nos entra un Dto en el caso de butacas existen unos validadores previos para comprobar la entrada de la butaca. En este caso tenemos ciertas etiquetas validadoras que podemos establecer contra los atributos.

```
data class ButacaCreateDto(  
    @Min(value = 1, message = "La fila no puede ser negativo ni ser menor que 0")  
    val fila: Int,  
    @Min(value = 1, message = "El numero de la butaca no puede ser negativo ni ser menor que 0")  
    val numero: Int,  
    @Min(value = 1, message = "El numero de la sala no puede ser negativo ni ser menor que 0")  
    val salaId: Int  
)
```

Donde cada entrada tendrá sus respectivos validadores previos a la entrada de los datos a la API.

ResultsError

En este caso tenemos los results error, vemos que básicamente es una sealed class con un mensaje que entra como parámetro el cual mapea las clases notFound y BadRequest.

```
sealed class ButacasError(val message: String) {  
    class NotFound(message: String) : ButacasError(message)  
    class BadRequest(message: String) : ButacasError(message)  
}
```

Exceptions

Un caso para analizar es el tema de las excepciones las cuales nos permite devolver errores pertinentes en este caso como ejemplo tenemos el UsuarioException el cual mapeamos dos status: NOT_FOUND y BAD_REQUEST.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class UsuarioNotFoundException(message: String) : RuntimeException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class UsuarioBadRequestException(message: String) : RuntimeException(message)
```

Mappers

Aquí expongo el tema de los mapeos de los Dtos entrantes y salientes dentro de la API en este caso presento el mapper de entradas el cual de forma interesante le entra una reserva la cual busca primeramente si existe y esa reserva la busca y dependiendo de esa reserva crea la entrada con esa reserva creada.

```
fun Entradas.toDto(reserva: Reserva) = EntradaDto(
    uuid = this.uuid,
    tituloPelicula = this.tituloPelicula,
    fechaDia = this.fechaDia,
    sala = this.sala,
    reservald = reserva.id!!,
    precio = this.precio,
    qr = this.qr
)
```

También existen modelos de mapeo a modelos de forma directa en este caso tenemos el de reservaDto.toModel el cual mapea los dtos a modelos propios de la base de datos.

```
fun ReservaDto.toModel() = Reserva(
    fila = this.fila,
    numeroButaca = this.numeroButaca,
    fechaHoraReserva = this.fechaHoraReserva
)
```

Modelos

En este caso tenemos el apartado de modelos los cuales son las entidades que mapearemos para el transcurso de entrada y salida de la base de datos en este caso como ejemplo tenemos butacas los cuales cuentan con ciertas etiquetas tales como:

@Table: Argumento el cual indica el nombre de la tabla expuesta.

@Id: Crea los id's de forma automática dentro de la base de datos.

@Column: Las columnas pertinentes de la base de datos.

```
@Table(name = "butacas")
data class Butacas(
    @Id
    val id :Int?=null,
    @Column("uuid")
    var uuid: String = UUID.randomUUID().toString(),
    @Column("fila")
    val fila:Int,
    @Column("numero")
    val numero: Int,
    @Column("salalid")
    val salalid: Int
```

Estos datos se encuentran dentro del archivo sql el cual muestra que han sido mapeada a la base de datos.

```
CREATE TABLE IF NOT EXISTS butacas(
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    uuid varchar(255) UNIQUE,
    fila INTEGER NOT NULL,
    numero INTEGER NOT NULL,
    salalid INTEGER NOT NULL
);
```


Repositories

Aquí expondremos un apartado importante el cual es los repositorios los cuales hacen las operaciones pertinentes sobre la base de datos CRUD y métodos varios que no están explícitamente impuestos dentro de los métodos normales de la clase.

En consecuencia, tenemos como implementación un `CoroutineCrudRepository` al usar coroutines, podemos realizar operaciones de acceso a datos de forma asíncrona, lo que puede mejorar el rendimiento y la capacidad de respuesta de tu aplicación. Esto es especialmente útil cuando tienes operaciones de E/S costosas, como acceder a una base de datos o llamar a una API externa.

También ya que estoy utilizando un enfoque reactivo en aplicación, como R2DBC en este caso, las coroutines son una buena opción para manejar operaciones asíncronas de forma reactiva. Podemos combinar la programación reactiva y las coroutines para crear un flujo de trabajo eficiente y reactivo.



En este caso tenemos como ejemplo el repositorio de butacas donde tenemos una relación 1-N la cual se maneja mediante el método de `findSalalId` donde ese método según las butacas creadas en esta sala nos devuelve un flujo con las butacas pertinentes añadidas a esa sala.

```
@Repository
interface ButacasRepository : CoroutineCrudRepository<Butacas, Int> {
    fun findByUuid(Uuid: UUID): Flow<Butacas>
    fun findBySalalId(salalId: Int): Flow<Butacas>
}
```

Interfaz Flows Con Results

En este caso tenemos como interfaz un conjunto de métodos los cuales vamos a modelar accediendo a un servicio el cual implementará todos estos métodos donde en algunos tendremos como objeto resultante un Result y su pertinente clase de error en este caso ButacasError.

```
interface ButacasServiceFlows {  
    suspend fun findAll(): Flow<Butacas>  
    suspend fun findByUuid(uuid: UUID): Result<Butacas, ButacasError>  
    suspend fun findById(id: Int?): Result<Butacas, ButacasError>  
    suspend fun findBySalalId(salalId: Int): Flow<Butacas>  
    suspend fun save(Butacas: Butacas): Result<Butacas, ButacasError>  
    suspend fun update(uuid: UUID, Butacas: Butacas): Result<Butacas, ButacasError>  
    suspend fun deleteById(id: Int): Result<Butacas, ButacasError>  
}
```

Algunos métodos de nuestro servicio son los siguientes:

```
override suspend fun findBySalalId(salalId: Int): Flow<Butacas> {  
    return repository.findBySalalId(salalId)  
}
```

```
override suspend fun save(Butacas: Butacas): Result<Butacas, ButacasError> {  
    return repository.save(Butacas).let { Ok(it) }  
}  
  
override suspend fun update(uuid: UUID, Butacas: Butacas): Result<Butacas, ButacasError> {  
    return findByUuid(uuid).onSuccess { it: Butacas  
        val updated = it.copy(  
            id = it.id,  
            uuid = it.uuid,  
            fila = Butacas.fila,  
            numero = Butacas.numero  
        )  
        return Ok(repository.save(updated))  
    }.onFailure { it: ButacasError  
        Err(ButacasError.NotFound("No existe la butaca con id: $uuid"))  
    }  
}
```

Validators

Aquí tenemos como clase de validación sobre los dto's de en este caso las butacas, tenemos un método de extensión sobre los dto's entrantes los cuales en caso de errores devolverán un result de forma errónea y en caso de pasar por cada argumento tendremos el objeto devuelto de forma positiva.

```
fun ButacaCreateDto.validate(): Result<ButacaCreateDto, ButacasError> {  
    if (this.fila < 1) {  
        return Err(ButacasError.BadRequest("La fila no puede ser menor que 1"))  
    } else if (this.numero < 1) {  
        return Err(ButacasError.BadRequest("El numero no puede ser menor que 1"))  
    }  
    return Ok(value: this)  
}
```

Properties Back

En este caso tenemos un archivo llamado **application.properties** el cual nos permite configurar apartados externos los cuales aplican sobre la aplicación los cuales pueden ser: La conexión con la BBDD, la creación y tiempo de expiración de los tokens jwt, ssl etc..

```
api.path=api  
spring.r2dbc.url=r2dbc:mariadb://127.0.0.1:3307/JFilms  
spring.r2dbc.username=admin  
spring.r2dbc.password=1234
```

```
server.ssl.key-store-type=PKCS12  
server.ssl.key-store=classpath:cert/JFilms_keystore  
server.ssl.key-store-password=1234567  
server.ssl.key-alias=JFilmsKeyPair  
server.ssl.enabled=false
```

6 IMPLEMENTACIÓN FRONT-END

En este apartado voy a relatar mi experiencia realizando el apartado gráfico de mi aplicación ya que de primeras es una nueva experiencia para mi ya que no había tocado este apartado con la soltura y dedicación implicada en este proyecto con lo cual ha sido un reto para mi el cual me llena de orgullo exponerlo.

Como primer punto expondré un pequeño resumen de lo que es angular. Angular es un framework de desarrollo web de código abierto creado por Google. Permite construir aplicaciones web de una sola página (SPA) de manera eficiente y estructurada. Utiliza TypeScript como lenguaje de programación y sigue el patrón de arquitectura Modelo-Vista-Controlador (MVC).

Una vez entrado en contexto ahora veremos la estructura de un proyecto en angular:

Carpeta "src": Contiene todo el código fuente de la aplicación.

- Carpeta "**app**": Aquí se encuentran los componentes, servicios y otros elementos específicos de la aplicación.
- Carpeta "**components**": Contiene los componentes de la aplicación.
- Carpeta "**services**": Aquí se encuentran los servicios utilizados para la lógica y la comunicación con el servidor.
- Carpeta "**models**": Contiene los modelos de datos utilizados en la aplicación.
- Carpeta "**assets**": Aquí se almacenan archivos como imágenes, archivos CSS, etc.
- Carpeta "**environments**": Contiene los archivos de configuración para diferentes entornos (desarrollo, producción, etc.).
- Archivo "**index.html**": Es el punto de entrada de la aplicación web.
- Archivo "**main.ts**": Es el archivo principal que inicia la aplicación.

Otros archivos:

- Archivo "**angular.json**": Contiene la configuración global del proyecto de Angular.
- Archivo "**tsconfig.json**": Es el archivo de configuración de TypeScript.
- Archivo "**package.json**": Especifica las dependencias y scripts utilizados en el proyecto.
- Archivo "**app.module.ts**": Define el módulo principal de la aplicación.

Partes Para Destacar

En este apartado explicaremos dos partes importantes dentro de mi proyecto de angular las cuales son: El Router y el método de inyección de componentes dentro de angular.

Router o ActivatedRoute



ActivatedRoute y el enrutamiento en Angular trabajan en conjunto para permitir la navegación entre componentes y proporcionar acceso a la información de la ruta actual en los componentes asociados. Esto es fundamental para construir aplicaciones web dinámicas y personalizadas.

El enrutamiento permite cargar componentes específicos en función de la URL actual, lo que proporciona una experiencia de navegación fluida.

ActivatedRoute es una clase proporcionada por el enrutador de Angular. Contiene información sobre la ruta activa, como los parámetros de la URL, los datos de la ruta, etc.

Aquí algunos ejemplos:

Router

```
onRegister(){  
  console.log("register");  
  this.router.navigate(['/registrarse']);  
}
```

ActivatedRoute

```
private route: ActivatedRoute,  
  
{  
  const listaReservadasOriginal = this.route.snapshot.queryParamMap.get('lista');
```

En el caso del Router es utilizado para navegar entre rutas de forma eficiente, mientras que en el caso de ActivatedRoute nos permite consumir esos parámetros que se les pueden pasar a las rutas mediante el Router.

Inyecciones en Angular

En Angular, la inyección de dependencias es un patrón de diseño fundamental que se utiliza para proporcionar y objetos y/o servicios en una aplicación. La inyección de dependencias permite crear componentes y servicios sin tener que preocuparse por crear manualmente todas sus dependencias. La inyección de dependencias se realiza a través del sistema de inyección de dependencias incorporado.

Cuando se necesita una dependencia en un componente o servicio, se especifica en el constructor del componente utilizando la inyección de dependencias facilita la creación de aplicaciones modulares, reutilizables y testeables, ya que reduce el acoplamiento entre los componentes.

Aquí Ejemplos:

```
constructor(  
  private location: Location,  
  private usuarioService: UsuarioService,  
  private reservaService: ReservaService,  
  private route: ActivatedRoute,  
) {
```

```
constructor(  
  private formBuilder: FormBuilder,  
  private router: Router,  
  private loginService: LoginService  
) {
```

Consumición del Back-End-API

Aquí expondré la creación de los servicios los cuales ayudan hoy a la comunicación del frontend con el backend ya que estos servicios con la capacidad de una clase llamada HTTP Client hoy son capaces de comunicarse mediante peticiones como get post updates y deletes ejecutar las acciones pertinentes expuestas a la API del back.



Dichos métodos están implementados con la capacidad de devolver un observable del objeto ya que se queda escuchando hoy sí hay algún cambio o hoy se ejerce algún tipo de error en la petición para ello se suscriben al método y según la respuesta que obtenga la imprimirá y en caso de error devolverá dicho error.

Aquí varios ejemplos:

```
export class LoginService {  
  constructor(private http: HttpClient) {  
  }  
  
  login(name: string, password: string){  
    return this.http.post<User>('http://localhost:7070/api/users/login', {name, password})  
  }  
  
  register(name: string, password: string, email: string, dni: string, tarjeta: string){  
    return this.http.post<User>('http://localhost:7070/api/users/register', {email, name, password, dni, tarjeta})  
  }  
}
```

```
export class UsuarioService {  
  constructor(private http : HttpClient) { }  
  
  getUsuario(username : string):Observable<Usuario>{  
    return this.http.get<Usuario>(`http://localhost:7070/api/users/${username}`)  
  }  
  
  updateUsuario(usuario:Usuario):Observable<Usuario>{  
    return this.http.put<Usuario>(`http://localhost:7070/api/users/me/${usuario.uuid}`, usuario)  
  }  
}
```

```
getCartelera():Observable<Movie[]>|any{  
  if(this.cargando){  
    return;  
  }  
  this.cargando = true;  
  return this.http.get<Movie[]>(`http://localhost:7070/api/peliculas`  
  ).pipe(  
    tap(() =>{  
      this.cargando = false;  
    })  
  )  
}  
  
getPeliculaDetalle(id:number){  
  return this.http.get<Movie>(`http://localhost:7070/api/peliculas/${id}`)  
  .pipe(  
    catchError(err => of(null))  
  )  
}
```

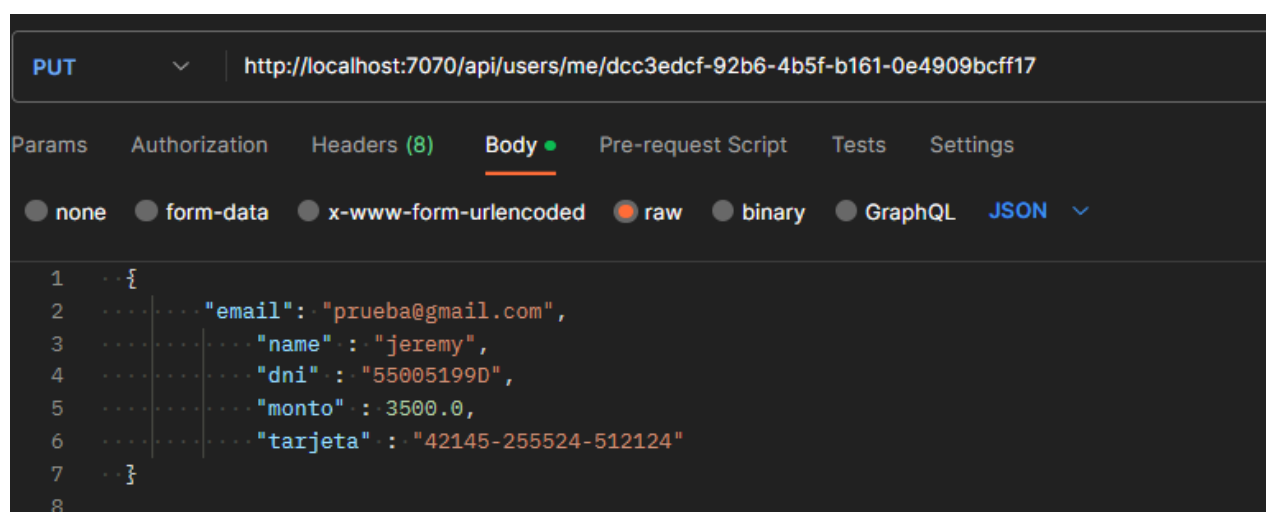
7 RESULTADOS Y DISCUSION

El resultado de la aplicación a mi sorpresa es mejor de lo que me esperaba ya que en apartado back como front me he empeñado a que sea a mi visión y mi parecer de las cosas y de una forma satisfactoria he podido concluir con lo que en mi cabeza tenía pensado.

Back-End

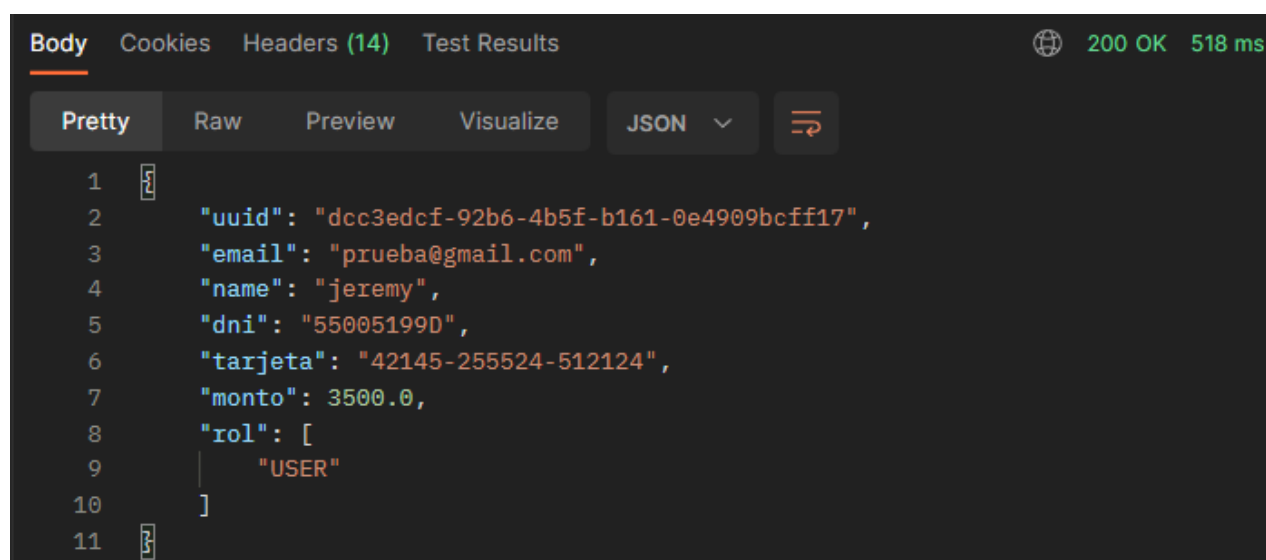
Aquí expondré varias capturas del postman ya que son los resultados de las distintas llamadas al back (API).

Un ejemplo es a la hora de actualizar un usuario.



A screenshot of the Postman application showing a PUT request. The URL is `http://localhost:7070/api/users/me/dcc3edcf-92b6-4b5f-b161-0e4909bcff17`. The request body is in JSON format and contains the following data:

```
1 {
2   "email": "prueba@gmail.com",
3   "name": "jeremy",
4   "dni": "55005199D",
5   "monto": 3500.0,
6   "tarjeta": "42145-255524-512124"
7 }
```



A screenshot of the Postman application showing the response of the PUT request. The status is 200 OK and the response time is 518 ms. The response body is in JSON format and contains the following data:

```
1 {
2   "uuid": "dcc3edcf-92b6-4b5f-b161-0e4909bcff17",
3   "email": "prueba@gmail.com",
4   "name": "jeremy",
5   "dni": "55005199D",
6   "tarjeta": "42145-255524-512124",
7   "monto": 3500.0,
8   "rol": [
9     "USER"
10  ]
11 }
```


Otro ejemplo es el apartado del login y la devuelta del usuario con token.

```
POST http://localhost:7070/api/users/login

Body
none form-data x-www-form-urlencoded raw binary GraphQL JSON
{
  "name": "jeremy",
  "password": "prueba01"
}
```

```
Body Cookies Headers (14) Test Results 200 OK 130 ms 952 B Save as Example
Pretty Raw Preview Visualize JSON
{
  "user": {
    "uuid": "dcc3edcf-92b6-4b5f-b161-0e4909bcff17",
    "email": "prueba@gmail.com",
    "name": "jeremy",
    "dni": "55005199D",
    "tarjeta": "42145-255524-512124",
    "monto": 3500.0,
    "rol": [
      "USER"
    ]
  },
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJhdWQiOiJqd3QtYXVkaWVuY2UiLCJzdWIiOiJkY2MzZWZrZiZi05MmI2LTRiNWYtYjE2MS0wZTQ5MDliY2ZmMTciLCJyb2xlcYi6IltVU0VSXSIsIm5hbWUiOiJqZXBjbXkiLCJleHAiOiJlY20yODY3NzQsIm1hdCI6MTY4NjQ4MzE3NCwiZW1haWwiOiJwcnVlYmFAZ21haWwY29tIn0.2UYPM45M_6dRTY_S93KZH6bWkeN00JiPXsLvol4A0s-1cehSqAf60qwx6-znoe4iULFc4Pt7aZuoyHEEpC53w"
```

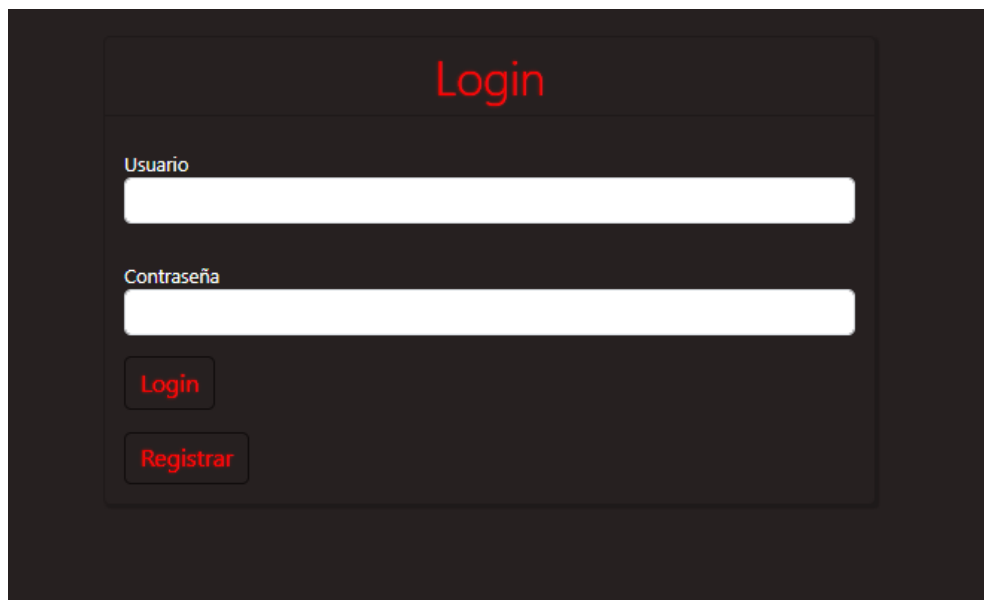
```
GET http://localhost:7070/api/users/jeremy
```

```
Body Cookies Headers (14) Test Results 200 OK 20 ms
Pretty Raw Preview Visualize JSON
{
  "uuid": "dcc3edcf-92b6-4b5f-b161-0e4909bcff17",
  "email": "prueba@gmail.com",
  "name": "jeremy",
  "dni": "55005199D",
  "tarjeta": "42145-255524-512124",
  "monto": 3500.0,
  "rol": [
    "USER"
  ]
}
```

Front-End

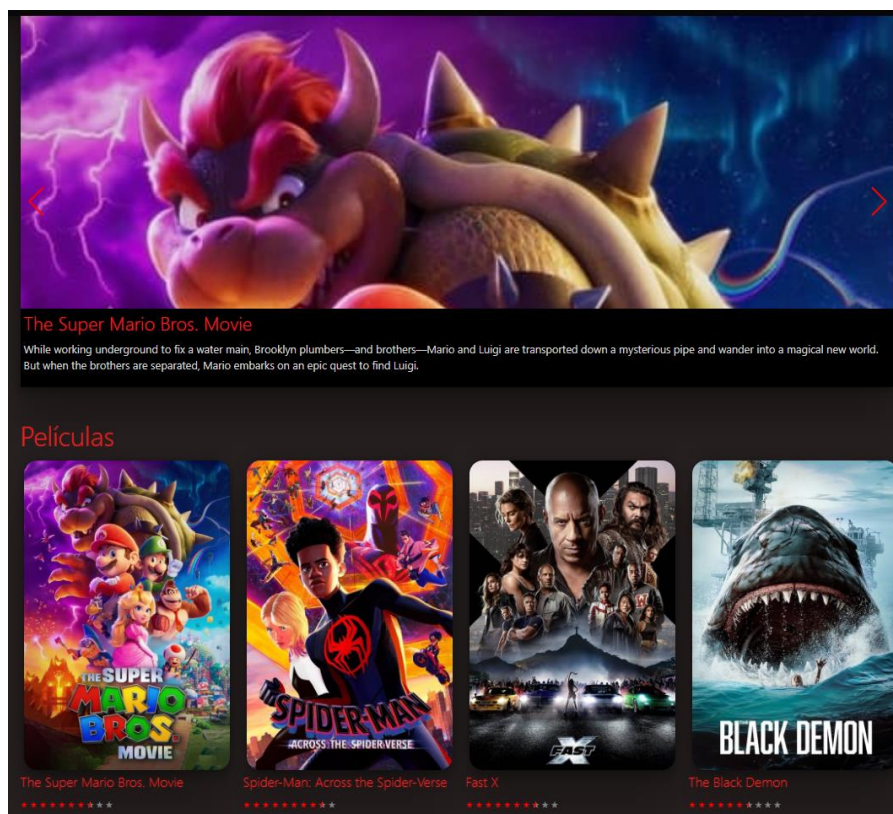
Aquí expondré distintas capturas del resultado de la realización del front de **JFILMS**.

Login




The login form is displayed on a dark background. It features a red 'Login' title at the top. Below the title are two white input fields: 'Usuario' and 'Contraseña'. At the bottom of the form are two buttons: 'Login' and 'Registrar', both with red text on a dark background.

Página Principal



Pantalla Película Elegida

[Volver](#)



★★★★★☆☆

7.8

The Super Mario Bros. Movie

While working underground to fix a water main, Brooklyn plumbers—and brothers—Mario and Luigi are transported down a mysterious pipe and wander into a magical new world. But when the brothers are separated, Mario embarks on an epic quest to find Luigi.

Salas

Sala 1
19:30

Sala 2
20:00

Sala 3
20:30

Pantalla Cine

+

Entrada Normal

9.99€

0

Pantalla

1234567

891011121314

15161719

[Reservar](#)

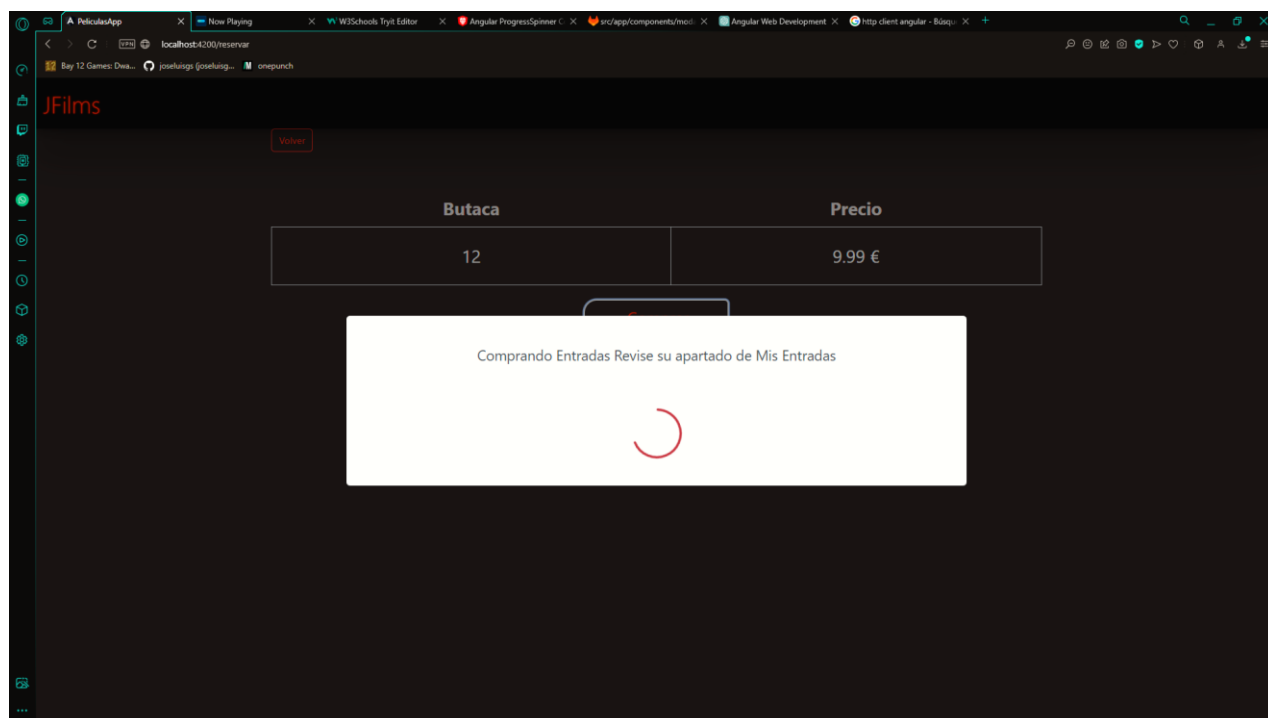
Pantalla Resumen

Volver

Butaca	Precio
12	9.99 €

Comprar
9.99 €

Creando Compra



Comprando Entradas Revise su apartado de Mis Entradas

Aspectos Varios

Costes

Hay que tener en cuenta que En España, los desarrolladores de software con experiencia junior suelen tener salarios por hora que oscilan entre los 15 y los 25 euros. Los desarrolladores más experimentados o senior pueden llegar a ganar entre 25 y 50 euros por hora. Sin embargo, estos números son solo estimaciones y pueden variar dependiendo de la ubicación y otros factores.

En este caso hemos presupuestado esta aplicación con un solo desarrollador senior planteamos que en 3 días tendría la aplicación con lo cual su presupuesto es de : 1800€

8 BLIOGRAFÍA

Angular. (s. f.). Angular.io. Recuperado 11 de junio de 2023, de <https://angular.io/docs>

Getting Started. (2012). En Having Success with NSF (pp. 1-16). John Wiley & Sons, Inc.

Get started with bootstrap. Getbootstrap.com. Recuperado 1 de mayo de 2023, de <https://getbootstrap.com/docs/5.3/getting-started/introduction/>

PrimeNG - Angular UI component library. (s. f.). Primeng.org. Recuperado 1 de mayo de 2023, de <https://primeng.org>

(S. f.). **Baeldung.com.** Recuperado 1 de mayo de 2023, de <https://www.baeldung.com/kotlin/kotlin-overview>