Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Estudio de la arquitectura de Microservicios y Spring Boot

Autor: Samuel García González

Tutor/es: Salvador Otón Tortosa

2020

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

ESTUDIO DE LA ARQUITECTURA DE MICROSERVICIOS Y SPRING BOOT

Samuel García González

10 / 2020

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Carrera

ESTUDIO DE LA ARQUITECTURA DE MICROSERVICIOS Y SPRING BOOT

Autor: **Samuel García González**
Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: _____

Vocal 1º: _____

Vocal 2º: _____

Calificación: _____

Alcalá de Henares a de de 2020

Para Patricia, por apoyarme incondicionalmente en el desarrollo de este trabajo y en toda mi vida, enseñándome que siempre podemos ser mejores.

Índice

Resumen

En Estados Unidos, uno de cada tres megabytes de todo su tráfico de Internet corresponde a Netflix. Que una sola aplicación pueda gestionar tanto tráfico es una idea que hace relativamente pocos años podría parecer hasta una utopía. La clave de su éxito se debe a su arquitectura, que está basada en los microservicios.

En este trabajo vamos a analizar la arquitectura monolítica, que es la manera clásica de construir aplicaciones y los motivos que existen detrás de la reciente popularización de la arquitectura de microservicios, con su respectivo análisis en detalle. En conjunto con esta introducción teórica, analizaremos el framework Spring Boot, ya que su papel en el desarrollo de aplicaciones Java basadas en la arquitectura de microservicios es fundamental hoy en día.

Una vez dispongamos de una buena base teórica, desarrollaremos una aplicación basada en esta arquitectura. Esta aplicación será un catálogo online de películas y series, en la que los usuarios podrán ver las características, el tráiler y las reseñas sobre los títulos que ofrece el catálogo, también teniendo la posibilidad de generar sus propias reseñas. El objetivo con el que desarrollamos esta aplicación es poder materializar todos los conceptos teóricos detrás de la arquitectura de microservicios y facilitar su comprensión.

Abstract

In the United States, one in three megabytes of all of its Internet traffic corresponds to Netflix. That a single application can handle so much traffic is an idea that relatively few years ago could seem like an utopia. The key to its success is due to its architecture, which is based on microservices.

In this project we are going to analyze monolithic architecture, which is the classic way of building applications, and the reasons behind the recent popularization of microservices architecture, with its respective analysis in detail. In conjunction with this theoretical introduction, we will analyze the Spring Boot framework, since its role in the development of Java based applications on the microservices architecture is fundamental at the present time.

Once we have a good theoretical base, we will develop an application based on this architecture. This application will be an online catalog of movies and series, in which users will be able to see the characteristics, the trailer and the reviews of the titles offered by the catalog, also having the possibility of generating their own reviews. The objective with which we develop this application is to be able to materialize all the theoretical concepts behind the microservices architecture and facilitate its understanding.

Palabras clave

Arquitectura de Microservicios, Arquitectura monolítica, Spring Boot y Spring Cloud Netflix.

Keywords

Microservices Architecture, Monolithic architecture, Spring Boot and Spring Cloud Netflix.

Introducción

Motivaciones personales

Este proyecto está motivado por la curiosidad en tecnologías actuales que hacen funcionar aplicaciones mundialmente conocidas como Netflix o Twitter. Me considero un chico bastante curioso y me fascinan los casos en los que conceptos simples a primera vista, ayudan a crear innovaciones en el mundo de la tecnología, como lo es el caso de los microservicios. Antes de realizar este proyecto no tenía casi conocimientos sobre la arquitectura, sólo había escuchado algunos casos en los que esta arquitectura se había aplicado con éxito, como los anteriormente mencionados.

La intención con la que elegí este tema es porque consideraba que necesitaba formación en esta tecnología en auge. Además, quería aprender a utilizar el framework de Spring y concretamente Spring Boot, dado que es el estándar de facto en el mercado para el desarrollo de aplicaciones Java. Por último, también me motivaba mucho el hecho de poder dar mis primeros pasos en el Full Stack Development de manera independiente y con mis ideas.

Después de repasar las motivaciones que han llevado al desarrollo de este proyecto, es hora de presentar el problema que se quiere resolver.

Planteamiento del problema

En este proyecto se quiere desarrollar una aplicación basada en microservicios que funcione como un catálogo de películas y series. Será una aplicación web con un diseño sencillo, que permita ver en funcionamiento la arquitectura de una aplicación basada en microservicios. Para hacer esto, en primer lugar, se presentará un análisis teórico extenso sobre la arquitectura basada en microservicios. La segunda parte del proyecto se centrará en documentar el proceso de desarrollo y explicar los conceptos, decisiones y tecnologías más importantes para realizarlo.

Objetivos Principales

1. Comprender qué es un microservicio.
2. Conocer la arquitectura basada en microservicios, sus ventajas y desventajas.
3. Conocer qué características nos ofrece y cómo funciona el framework Spring Boot.
4. Construir una aplicación basada en la arquitectura de microservicios.

Objetivos secundarios

1. Aprender a gestionar la comunicación entre microservicios.
2. Desarrollar una capa de presentación atractiva para el usuario.
3. Seguir buenas prácticas y estructurar las aplicaciones según patrones definidos.

Aplicaciones monolíticas o tradicionales

Características de una aplicación monolítica o tradicional

El modelo o arquitectura tradicional de aplicaciones se caracteriza porque las aplicaciones son presentadas como artefactos software únicos y desplegables. Toda la Interfaz de Usuario (UI), lógica y bases de datos están empaquetadas en una única aplicación y desplegadas en un servidor dedicado a la misma.

Esta arquitectura suele estar formada por tres capas, la de presentación (front-end), la de lógica (back-end) y la de datos (BBDD).

También se conoce a esta arquitectura como "monolítica", porque es un producto pensado y desarrollado como una unidad. A pesar de esto, suelen trabajar en él numerosos grupos de desarrollo simultáneamente. Lo más habitual es que estos se repartan las diferentes funcionalidades o módulos del producto para desarrollarlas de la manera más paralela e independiente posible.

Este paralelismo es atractivo y ventajoso para los desarrolladores porque el desarrollo de los diferentes equipos avanza de manera independiente y van depositando el código desarrollado en una única base de código compartida.

Las bases de esta arquitectura son sencillas de comprender, pero trae consigo una serie de problemas y riesgos intrínsecos.

Riesgos y problemas la arquitectura monolítica o tradicional

El mayor problema de esta arquitectura surge cuando el equipo tiene que realizar cambios una vez la aplicación ya ha sido desplegada. Tanto como si añadimos una nueva característica o cambiamos algún fragmento de código, por pequeño que sea, para poder poner estos cambios en el entorno de producción, la aplicación entera tiene que volver a ser recompilada, retesteada y redesplegada. Cuanto más grande sea la aplicación, más tardará este proceso.

Además de los costes temporales, causa la interrupción de los servicios de nuestra aplicación y si añadimos los riesgos por ser un proceso en el que, si hay algún fallo, la aplicación entera cae, es lógico pensar que el proceso de despliegue es el más crítico de toda la aplicación.

Otro factor que suele perjudicar a las aplicaciones monolíticas es la dificultad para escalarlas. Es una tarea muy costosa y complicada, porque la demanda de los servicios que ofrece la aplicación es desigual por naturaleza. Llegado el caso en el que necesitemos escalar uno en concreto para que coincida con el aumento de demanda,

estaremos obligados a escalar el conjunto de la aplicación al mismo tiempo. Esto causa que el coste de escalada no sea efectivo, puesto que escalaremos partes de la aplicación que no lo necesitan, y además requiera de una cantidad de recursos muy grande para llevarla a cabo.

Por otra parte, su mantenimiento y prevención de la degradación del software es una tarea con una complejidad elevada, que aumenta también junto con el tamaño de la aplicación. Buscar el origen de un pequeño bug puede requerir una búsqueda exhaustiva entre las clases de la aplicación. Al sufrir de un fuerte acoplamiento entre clases y librerías, podemos encontrarnos en la situación en la que al actualizar una sola de ellas puede resultar en que la aplicación acabe fallando o teniendo comportamientos inesperados.

Por naturaleza las aplicaciones tradicionales manejan multitud de diferentes tipos de datos. Estos suelen estar en una base de datos única de la que dispone la aplicación, lo que significa que los desarrolladores pueden verse en el caso en el que accedan a datos que no corresponden al dominio en el que están trabajando directamente. Esta situación crea dependencias ocultas y puede que un solo cambio en la base de datos requiera una propagación de cambios a multitud de código en toda la aplicación.

En cuanto al desarrollo de aplicaciones monolíticas muy grandes, este proceso puede no ser óptimo debido a conflictos entre los requisitos de los diferentes departamentos o descoordinaciones en las entregas.

Por último, es importante tener en cuenta que la elección de lenguaje y framework acompañará a esta aplicación durante todo su ciclo de vida. Esto sucede porque para cambiarlos tendríamos que reconstruir la aplicación entera prácticamente debido a las dependencias que generan en toda ella.

Aplicaciones basadas en microservicios

Por qué cambiamos la manera de construir aplicaciones

Desde la aparición de Internet en 1969, esta red de redes ha transformado e interconectado todos los aspectos de la sociedad. Desde un punto de vista del mercado, se benefician tanto las multinacionales como empresas locales, ya que ambas pueden distribuir su producto a nivel global, aunque también trae consigo ciertas "desventajas" ya que al aumentar tanto la base de clientes potenciales aumenta el número de competidores, ya que todas las empresas pertenecen al mismo mercado globalizado. Esta presión por sobrevivir en la lucha competitiva ha afectado a la manera en la que los desarrolladores construyen las aplicaciones. Podemos identificar varios factores que empujaron al mundo del desarrollo a crear la arquitectura basada en microservicios:

- Complejidad: Las empresas actualmente suelen externalizar servicios, los cuáles antes estaban incluidos en su propia estructura. Este cambio se ve reflejado en que las aplicaciones monolíticas para gestionar la empresa se hayan visto sustituidas y surjan aplicaciones que se comuniquen con varios servicios y bases de datos, que pueden pertenecer a la empresa o no.
- Rapidez: Hoy en día los clientes habitualmente se decantan por productos software sean actualizados en intervalos de tiempo cortos o medios, para disponer de nuevas características cuanto antes.
- Rendimiento y escalabilidad: Al pertenecer a un mercado global, las aplicaciones deben tener mecanismos para escalar y desescalar según la demanda y mantener el rendimiento en todas las situaciones, ya que no sabemos exactamente cuál puede ser el volumen de transacciones en un momento determinado.
- Fiabilidad: Nuestra aplicación debe ser altamente fiable porque un sólo problema puede hacer que nuestro cliente abandone nuestro servicio y esto empeore nuestra imagen en el mercado.

El concepto fundamental detrás de los microservicios es, aunque sea paradójico, que los desarrolladores debemos comprender que para construir servicios altamente escalables para poder dar servicio a una mayor cantidad de clientes necesitamos desgranar nuestras aplicaciones en pequeños servicios que puedan ser construidos y desplegados de manera independiente. Este concepto nos permite construir sistemas que son:

- Flexibles: Dado que nuestra aplicación se compone de servicios pequeños y de única función, podemos reemplazarlos o cambiarlos para ofrecer nuevas funcionalidades rápidamente.
- Resilientes: Esta arquitectura tiene una gran resistencia ante los fallos gracias a la modularidad del sistema. Esto nos permite que cuando un servicio falle, pueda ser sustituido rápidamente por otro. También será resistente a la degradación del código, debida al paso del tiempo y al avance en las tecnologías que rodean

a una aplicación. Esta se suaviza en la arquitectura de microservicios, ya que, al igual que con los fallos, la modularidad de la aplicación nos permite ir actualizando y reemplazando los servicios obsoletos sin necesidad de reconstruir la aplicación a gran escala.

- Escalables: Nuestra aplicación se debe poder escalar horizontalmente a través de diferentes servidores. Esto nos permite responder a los aumentos de la demanda de cualquier servicio de la aplicación, haciendo ese coste efectivo ya que únicamente potenciamos la disponibilidad del recurso necesario.

Definición y principios de los microservicios

Los microservicios nos permiten desarrollar aplicaciones con módulos físicamente separados. Amazon, Netflix y eBay son ejemplos de aplicación exitosa de esta arquitectura. Surgieron basados en la idea propuesta por Alister Cockburn en 2005 de la arquitectura hexagonal (también conocida como arquitectura de Puertos y Adaptadores). Esta arquitectura proponía encapsular las funciones de negocio del resto del mundo. Estas funciones, al estar aisladas, no conocían los canales de entrada o los formatos de mensajes que podían recibir. Esto se solucionaba convirtiendo los diferentes mensajes que recibía la aplicación desde diferentes dispositivos a uno que fuese conocido por las funciones. El proceso que llevan a cabo los puertos y adaptadores pasa desapercibido tanto por las aplicaciones externas como las funciones internas. Esto nos permite realizar cambios sin tener que preocuparnos demasiado por los diseños de interfaces.

Una analogía que nos hará entender mejor los microservicios es un panal. Las abejas construyen el panal juntando celdas hexagonales de cera. Empiezan poco a poco, construyendo este con los materiales que tienen disponibles para construirlo. Repiten el patrón de construcción de las celdas, creando una estructura muy robusta, que está compuesta por celdas individuales unidas unas con otras. Los daños a una sola celda no repercuten a las demás y se pueden reconstruir fácilmente

No existe una definición oficial sobre qué es un microservicio, pero la concepción sobre los mismos más aceptada es que es una técnica de desarrollo de aplicaciones software según la cuál podemos construir una aplicación como un conjunto de servicios modulares, pequeños e independientemente desplegables (¡Como las abejas en el panal!). Actúan como un sistema distribuido en el que cada servicio tiene deseablemente una única responsabilidad y se intercomunican a través de protocolos independientes de la implementación como JSON o HTTP.

En la popularización de la arquitectura de los microservicios contribuyó en gran medida el avance de tecnologías como HTML 5, Angular y la expansión de los proveedores de servicios cloud PaaS. Por último, no podemos olvidar que Docker dio el impulso final a esta arquitectura para que se lanzase de lleno a la popularidad. De estas cuestiones hablaremos más adelante, ahora vamos a repasar los principios en los que se fundamenta esta tecnología:

- Principio de única responsabilidad: Es uno de los principios de los patrones de diseño software SOLID, que están orientados al desarrollo de aplicaciones que usen la programación orientada a objetos (OOP). Describe que una unidad debe tener una única responsabilidad. Este principio abarca a funciones, clases o servicios, que actúan como unidad. Como podemos deducir, las responsabilidades no pueden ser compartidas entre servicios, aunque existen excepciones en las que por limitaciones del negocio podemos saltarnos este principio. La unidad en una aplicación basada en microservicios sería, lógicamente, un microservicio. Este principio nos aporta una alta cohesión en cada uno de los microservicios.

- Principio de Autonomía: Los microservicios son servicios autónomos e independientemente desplegables, que toman la responsabilidad completa sobre una tarea y su ejecución. Cuando un servicio es desplegado, decimos que se ha instanciado una copia de él. Pueden operar y cambiar independientemente del resto que le rodean. Recogen las dependencias como librerías, entornos de ejecución, e incluso a sí mismos en servidores, contenedores o maquinas virtuales para abstraer los recursos físicos.

- Principio de Descentralización: Las aplicaciones basadas en la arquitectura de microservicios poseen una gestión de datos descentralizada, ya que cada servicio gestiona su propia base de datos (si la necesita), dándonos mucha flexibilidad y consistencia para realizar cambios a múltiples recursos.

- Principio de bajo acoplamiento: Interactúan por interfaces claramente definidas o a través de eventos, mientras la implementación interna permanece independiente. Esto causa que los microservicios actúen como cajas negras, por lo que, desde fuera de ellos, no podemos conocer aspectos como sus estructuras de datos, tecnologías o su lógica. Esto también tiene una repercusión cultural en el mundo del desarrollo, pues permite que los equipos de desarrollo se organicen de manera independiente y la aplicación se construya rápidamente ensamblando los diferentes componentes.

- Principio de Resiliencia: Los microservicios son un mecanismo natural para aislar los fallos. Al ser independientes, si ocurre un fallo lo más probable es que ese fallo afecte sólo a una parte del sistema. También nos permite añadir cambios más pequeños a la aplicación en vez de actualizaciones muy grandes que pueden ser peligrosas.

- Principio de Antifragilidad: Recuperarse rápido de los fallos es indispensable para construir sistemas resistentes y tolerantes a ellos. Este principio asume que nuestra aplicación puede fallar (e inevitablemente ocurrirá), y cuando detecta un fallo debe recuperarse de él lo más rápido posible, eliminando los procesos que sean necesarios para no afectar al sistema completo. La auto sanación suele

acompañar a este principio y se usa habitualmente en microservicios, donde el sistema puede detectar los fallos y se ajusta automáticamente para evitar seguir fallando.

- Principio de bajo peso: Al implementar una función específica, estos son muy ligeros. Debemos tener en cuenta la selección de tecnologías que lo respaldan, asegurándonos que mantendrán esta característica. Esto permite que su instanciación y despliegue sean muy rápidos y baratos.

- Principio de reusabilidad: Al estar concebidos como cajas negras, podemos reutilizar microservicios ya codificados y probados ajenos a la aplicación que encajen con una parte de esta. Debemos tener en cuenta que la interfaz de comunicación y los resultados que ofrezca el microservicio estén en consonancia con nuestra aplicación. Este principio es muy abierto ya que, además, podemos reusar las partes que nos interesen de microservicios construidos.

- Principio de Interoperabilidad: Los microservicios conviven en un ecosistema donde existen variedad de ellos, y consiguen sus objetivos mediante la comunicación.

- Principio de transparencia: La monitorización de cada microservicio es muy importante para asegurar que cuando ocurran problemas, sabremos cuál es su fuente y poder resolverlos eficazmente.

- Principio plurilingüe: Los microservicios deben comunicarse entre ellos para conseguir sus objetivos. Esto se realiza con protocolos de comunicación independientes de la implementación, lo cual nos permite que los servicios puedan estar construidos con diferentes tecnologías y lenguajes.

- Principio de automatización: El ciclo de vida de los microservicios (que explicaremos más adelante) debe estar completamente automatizado ya que gestionarlo manualmente no tendría sentido por el alto coste y esfuerzo que conllevaría, sin ofrecernos prácticamente ninguna ventaja. Implementar esta característica es una tarea compleja, pero a la larga nos ahorrará tiempo y posibles fallos en varios procesos. Los procesos de build, testeo, despliegue y escalada son los que debemos automatizar para conseguir manejar el ciclo de vida de los servicios de esta manera.

- Principios del ecosistema de la aplicación: Alrededor de los servicios también tendremos que organizar un ecosistema para mejorar la calidad de la aplicación. Entre muchos aspectos, los más importantes son procesos DevOps, gestión de un log centralizado, API Gateway y monitorización. Estos conceptos se verán en más profundidad junto con la arquitectura de los microservicios.

Tras esta revisión de principios de los microservicios podemos entender por qué los microservicios supusieron una revolución en su momento. Para implementar todas ellas, debemos conocer a fondo la arquitectura y patrones típicos de una aplicación basada en microservicios, que son las que consiguen que los microservicios nos puedan ofrecer todas estas características.

Características de la arquitectura de una aplicación basada en microservicios

Como ocurría con la definición del concepto de microservicio, tampoco existe un modelo estándar para desarrollar una aplicación basada en microservicios. Aunque esta situación nos pueda hacer pensar que, al no tener una construcción teórica fija, los microservicios no son una tecnología fiable o segura, la realidad es que es todo lo contrario. Existen infinidad de patrones y variaciones para aplicar esta arquitectura. Es importante que tengamos en cuenta que para construir una aplicación basada en microservicios no tenemos exclusivamente que escribir código, sino que también debemos tener en cuenta diversos aspectos como cuál es el tamaño correcto de un microservicio, cómo gestionar la escalabilidad o los fallos de la aplicación. Al igual que con la definición, vamos a intentar definir una serie de características comunes a todas las arquitecturas basadas en microservicios.
En primer lugar, vamos a definir la estructura más típica de la arquitectura basada en microservicios:

La estructura habitual de una aplicación de microservicios es de cuatro capas. Vamos a definirlas y explorarlas:

- Plataforma: Los microservicios no viven aislados, se encuentran en una infraestructura compuesta por muchos servicios diferentes y herramientas que gestionan y automatizan procesos. Estas herramientas suponen un coste adicional a la lógica de negocio en si misma, pero nos ahorrarán en gran medida el esfuerzo para desplegar futuras versiones y actualizaciones. En esta capa destacamos los siguientes componentes:
    - Plataformas para ejecución como imágenes virtuales o contenedores.
    - Herramientas para ensamblar el log y monitorizar servicios.
    - Tuberías de despliegue consistentes para testear y desplegar nuevos servicios y versiones.
    - Canales de comunicación y agentes de localización de servicios.
- Servicios: En esta capa, como indica su nombre, se alojan los servicios. En ella interactúan los servicios para realizar su trabajo, dependiendo de la infraestructura inferior de la plataforma y comunicando su resultado a la capa de límites. En esta capa encontraremos:

- Lógica de negocio y de apoyo: Servicios que cumplen requisitos de negocio y los que aportan alguna funcionalidad a estos mismos.

- Servicios de ensamblaje: Manejan y transforman datos que proceden de múltiples servicios.

- Servicios en caminos críticos o no críticos: Servicios que dependen de los resultados de otro servicio se encuentran en un camino crítico.

- Comunicación: Pertenece a la capa de plataforma, pero una vez explicados los servicios merece la pena indagar en este apartado. Como transporte usaremos protocolos de comunicación independientes de la tecnología como lo es REST. Para establecer cómo nos comunicamos, existen dos patrones útiles en los microservicios:

    - Mensajes síncronos: Este tipo de comunicación es especialmente útil cuando queremos comunicar resultados de acciones o errores de ejecución. Viene acompañado de una serie de limitaciones, como que aumenta el acoplamiento entre servicios, limitan el trabajo en paralelo y bloquean código esperando a respuestas.

    - Mensajes asíncronos: La mejor manera para aprovechar este tipo de comunicación es con eventos. Permite una evolución fluida ya que los eventos los consumirán servicios que estén libres, permitiendo ajustarse a las circunstancias sin afectar a los servicios que se encuentran trabajando. Requiere de un agente de comunicación para gestionar los eventos que recibe y posteriormente mandarlos. Existen dos patrones de comunicación asíncrona: Cola de trabajos y publicador consumidor.

- Límites: Esta capa se encuentra entre los servicios y el cliente y oculta la complejidad interna de la aplicación y proporciona el acceso a la misma. En esta capa podríamos encontrar otras funcionalidades como:

    - Autenticación y autorización.

    - Recolección de métricas y ensamblaje del log.

    - Limitación de la ratio de acceso, para evitar abusos por parte de clientes.

    - API gateways proporcionan un punto de entrada único para el cliente hacia un servicio del backend. Se encarga de autenticar peticiones de clientes y gestionar la comunicación con el servicio y éste, pudiendo realizar también tareas adicionales como la autenticación o componer respuestas a partir de diferentes servicios.

- Clientes: Como la capa de presentación en la arquitectura de tres capas, esta se encarga de ofrecer una interfaz de usuario para la aplicación. Como está separada del backend, podemos adaptarla a multitud de usuarios, como navegadores, dispositivos móviles, etc. Esta interfaz proporciona acceso a todas las funcionalidades para satisfacer las necesidades de todos los tipos de usuarios. La estructura de la misma no es el objeto de estudio de este trabajo, pero podemos mencionar que existen front-ends monolíticos, pero en los últimos años se ha encontrado con las mismas dificultades que las aplicaciones monolíticas y están surgiendo los micro-frontends, soportados actualmente por Angular y muchos más frameworks.

Ahora procedemos a revisar una serie de patrones muy útiles, que extenderán conceptos presentados en las diferentes capas que hemos explicado y que encontraremos en la mayoría de las aplicaciones basadas en microservicios. Los dividiremos en seis aspectos clave siendo los siguientes:

- Patrones de desarrollo principales: Son patrones que definen las características básicas a la hora de construir un microservicio

    - Granularidad del servicio: La mejor manera para ajustar el tamaño de los microservicios es empezar siendo poco estricto, no refinando demasiado los servicios, y poco a poco encapsular funcionalidades que no sigan el principio de responsabilidad única según avanzamos en el desarrollo (aunque no hace falta ser extremadamente dogmático, como hemos mencionado en las características de los microservicios). Nuestro objetivo es identificar los enlaces más importantes entre microservicios y cómo llevarlos a cabo. Esto nos permite avanzar rápidamente y que nuestro proyecto madure adecuadamente. Además, cuando nuestra aplicación crezca y requiera nuevas funcionalidades, seguiremos este mismo proceso y refinaremos las ya existentes para implementarlas.

    - Protocolos de comunicación: A la hora de diseñar la comunicación con los servicios, debemos seguir la aproximación REST. Se recomienda usar URIs para los intentos de comunicación y JSON para las peticiones y respuestas. Finalmente, para comunicar los resultados, el protocolo HTTP es el estándar.

    - Procesamiento de eventos entre servicios: La comunicación de eventos entre servicios es crítica en el diseño de una buena aplicación basada en microservicios. Debe ser asíncrona y permite a la aplicación escalar mejor y superar los fallos. Se gestionan en cola y un servicio se encarga de procesarlos para conocer el estado de los servicios.

- Patrones de enrutamiento: Tratan sobre cómo podemos encontrar y comunicarnos con un microservicio que necesitamos.

    - Localización de servicio: En las aplicaciones basadas en microservicios se recomienda crear una capa de localización de servicio, por encima de las diferentes instancias de los servicios. Una instancia, ya mencionada anteriormente, es una copia del servicio que está desplegada y ejecutando sus funciones en la aplicación. Esta capa se encarga de conocer qué servicios están ejecutándose y operativos, cuál es su IP y su salud. Está compuesta por varios agentes que reciben información de la salud de todas las instancias de los microservicios Cuando se inicia un microservicio, este es registrado por la capa de localización, y se comprueba su salud mediante "latidos". Cuando una instancia deja de enviar latidos, se da por muerta y se elimina su dirección física. Esta capa mantiene una caché con las direcciones de servicios para facilitar la tarea de localización.

    - Enrutamiento de servicio: La aproximación recomendada es que todas las llamadas a los servicios sean gestionadas por la capa de localización, anteriormente mencionada. Será una puerta de entrada, con la que se comunicarán los servicios externos a la aplicación, y la capa de localización se encarga de encontrar el servicio solicitado. Esta capa debe ser muy ligera y apenas contener código, dado que podría ser un cuello de botella si no gestionamos bien la escalabilidad en esta capa. Por ello, debe tener mecanismos para asegurar el balance de carga.

- Patrones de resiliencia: Todos los sistemas experimentan fallos, especialmente los sistemas distribuidos, como lo son las aplicaciones basadas en microservicios. Estos patrones nos ayudan a gestionar estas situaciones.

    - Balance de carga: Como hemos mencionado antes, la capa de localización posee agentes que localizan las instancias de los servicios. En ellos debemos controlar el rendimiento de los servicios y redirigir la carga de los que estén saturados o rindiendo por debajo de lo esperado a los que estén libres. Este balanceador de carga también puede ignorar instancias si se considera necesario dado su rendimiento pobre o errores.

    - Patrón de cortocircuito: Este patrón nos permite cortar la conexión si al realizar una llamada a una instancia ésta tarda demasiado en responder. Cuando esto sucede, se corta la llamada a esa instancia y registrará ese fallo. Esta implementación nos permite evitar los servicios que por alguna razón estén dando fallos y evita hacerles llamadas, lo cuál nos ahorrará posibles errores futuros.

- Patrón de mampara: Surge basado en la razón por la que los barcos están divididos en partes que son estancas entre sí. Para que un servicio interactúe con varios recursos remotos y permita aplicar esa idea, lo recomendable es dividir los recursos en diferentes pools, las cuales contendrán los posibles fallos en ellas mismas y no lo propagarán, previniendo afectar a toda la aplicación.

- Patrones de seguridad: Nuestra aplicación debe proteger su funcionamiento y datos. Vamos a explorar los patrones más populares usados en microservicios.

    - Autenticación: El framework OAuth2 es un estándar en la industria para controlar la autenticación. Para implementar OAuth2 necesitamos crear un servicio que lo contenga. Ese servicio se encargará de registrar cada aplicación que quiera acceder a cualquiera de los servicios, le asignará un nombre y clave únicos. Se asegura que cada microservicio que ejecute una petición no tenga que presentar las credenciales con cada llamada.

    - Autorización: Cada servicio de nuestra aplicación se encargará de definir las acciones que un rol puede realizar. OAuth2 se encargará de validarlos.

- Patrones de registro en el log y de rastreo: El proceso de depuración en un sistema distribuido como lo son los microservicios es muy complejo, dada la existencia de múltiples servidores. Estos patrones nos ayudan a sobrellevar esa complicación y facilitarnos esa tarea tan necesaria.

    - Correlación del log: Cuando se inicia una transacción, que puede recorrer múltiples servicios, se asigna un ID que se irá propagando por cada servicio que recorra. Con esta identificación, podemos relacionar los eventos que se reflejen en el log con una transacción.

    - Ensamblaje del log: Existen multitud de soluciones, desde open source a productos comerciales. Mencionamos Elasticsearch, Papertrail o Splunk. Estos productos recogen todas las entradas de log e intentan identificar las entradas más relevantes del mismo.

    - Traza de microservicios: La solución más popular es Zipkin. Esta herramienta nos permite visualizar gráficamente las transacciones entre múltiples servicios y el tiempo transcurrido en cada uno de ellos.

- Patrones de despliegue: La idea principal detrás de estos es asegurar que nuestros microservicios se despliegan rápido y que las tareas de corrección de bugs o implementación de nuevas características también lo son.

- Tubería de build y despliegue: Es básicamente el patrón de Integración Continua/Entrega continua (CI/CD). Debemos establecer una tubería en la que, al subir nuevo código a nuestro repositorio, se haga una nueva "build" de la aplicación, se ejecuten tests unitarios y de integración y tras pasarlos, se ponga finalmente en funcionamiento en el entorno de producción. Esto nos otorga una enorme rapidez a la hora de desplegar el sistema, ya que es un proceso automatizado y seguro del que no tenemos que encargarnos haciéndolo manualmente.

- Código como infraestructura: Muy relacionado con el anterior patrón, debemos crear una infraestructura de despliegue automático, en la que las manos humanas no interfieran una vez la imagen de un microservicio se encuentre compilada y testeada. Este proceso se realiza con una serie de scripts que habremos creado previamente. Deben ser gestionados como una pieza de código igual de importante que el resto de la aplicación.

- Servidores inmutables: Este patrón nos evitará los errores de configuración típicos "porque al administrador se le ha ido la mano". Cada vez que se crea una nueva "build" de la imagen del servidor, esta no podrá modificarse. Si se requiere realizar alguna modificación, se modifican los scripts que provisionan al servidor y se creará una nueva "build" de este.

- Servidores fénix: Se basa en la idea de que las imágenes de servidores deben poder morir y reiniciarse sin afectar al comportamiento (cuando se vuelva a levantar) de los microservicios ni del servidor mismo. Nos da una gran autonomía y tranquilidad dado que nuestro servidor siempre estará online, aunque existan fallos y podremos identificar qué situaciones conducen a ellos.

Ventajas frente a la arquitectura monolítica

Ahora que tenemos un conocimiento extenso sobre las características de las aplicaciones basadas en la arquitectura de microservicios, podemos pasar a compararla con la arquitectura monolítica. En este apartado nos centraremos en los aspectos que mejora frente al monolito:

- Quizá la mejora más importante de todas es que permite construir sistemas orgánicos, que crecen con el tiempo y que este proceso se pueda hacer de una manera controlada. La ampliación de un servicio tendrá un impacto mínimo en el resto de los servicios, o directamente no tendrá alguno si no necesitamos modificar la interfaz de comunicación. Esto en la arquitectura monolítica sería imposible ya que probablemente tendríamos que realizar cambios en cascada

tras modificar un componente de la aplicación para que el funcionamiento de la aplicación sea correcto.

- Podemos realizar cambios con una gran independencia, ya que, simplemente manteniendo la interfaz de comunicación, podemos realizar cambios desde un pequeño fragmento de código a funcionalidades enteras o cambios de tecnologías a cualquier servicio que queramos sin afectar al resto de la aplicación con cambios en cascada, como sucedería en una aplicación monolítica. Además, esta característica favorece la innovación y la experimentación al contener los fallos aislados. Adicionalmente, ayuda a luchar contra la deuda tecnológica por código mal desarrollado, anticuado o en proceso de degradación. En vez de poner parches, reemplazamos esas partes por unas nuevas completamente funcionales y actualizadas.

- Cada microservicio posee sus propias estructuras de datos y fuentes de donde los obtiene. Esto implica que los datos que posee un microservicio sólo pueden ser modificados por ese mismo servicio. También podemos limitar el acceso a la base de datos y únicamente permitir a un servicio acceder a un tipo de datos. En las aplicaciones monolíticas solemos tener una única base de datos y esto, como hemos mencionado en los problemas de la arquitectura monolítica, puede conducir a problemas futuros.

- Al contrario que las aplicaciones tradicionales (o monolíticas) los microservicios pueden compilarse y desplegarse independientemente del resto de los que existen en la aplicación. Esto nos ahorra una gran cantidad de tiempo y facilita enormemente la tarea de testeo. Además, esto nos aporta seguridad y reduce el peligro de este proceso, ya que si el despliegue falla, la aplicación seguirá su ejecución sin caer entera, como sucede en las aplicaciones monolíticas.

- Coste efectivo en escalabilidad dado que únicamente se escalan los servicios que tienen un aumento en demanda y su rendimiento está por debajo de lo deseado. De esta manera el coste de escalar la aplicación es 100% efectivo, siendo esta efectividad imposible de conseguir en la arquitectura monolítica.

- Los límites entre el código están bien definidos porque esta arquitectura nos permite separar los diferentes dominios de la aplicación claramente, dado que cada servicio posee sus responsabilidades únicas definidas. A la hora de repartir el trabajo entre el equipo de desarrollo, cada equipo tendrá bien definidos sus objetivos, lo cual en el desarrollo de aplicaciones grandes es una ventaja muy notable. Este proceso en una aplicación monolítica es notablemente más complicado, por los problemas de definición de requisitos y coordinación de los equipos en las aplicaciones monolíticas.

- Esta arquitectura permite la coexistencia de diferentes versiones de un servicio cuando lo actualizamos, sin crear conflictos en el sistema. Esto en las arquitecturas monolíticas es imposible ya que sólo puede existir una versión desplegada al mismo tiempo.

- Al ser una arquitectura dirigida por eventos (tanto de entrada como de salida) y tener un gobierno descentralizado facilita la monitorización y la extracción de datos.

- Los microservicios van de la mano de DevOps, puesto que esta arquitectura tiene en su corazón este tipo de procesos, como build, testeo, despliegue y monitorización. Una de las ventajas frente a las aplicaciones monolíticas es que implementar la CI/CD en ellas es bastante complicado, pero en una aplicación basada en microservicios esta característica es natural.

Transición de una aplicación monolítica a la arquitectura basada en microservicios

En este apartado trataremos el proceso de transición de una aplicación monolítica a microservicios. Es una tarea que está a la orden del día ya que las empresas están interesadas en aplicaciones modulares que sustituyan a sus aplicaciones de gestión centrales monolíticas que, habitualmente, suelen ser anticuadas y con multitud de problemas acumulados.

Antes de empezar, tenemos que recordar que una aplicación monolítica está separada en capas horizontalmente, típicamente siendo tres: datos, lógica y presentación. Cada capa aporta servicios a su superior. Podemos establecer una cierta similitud entre un microservicio con un monolito, dado que contiene datos, lógica y devuelve datos a través de APIs (un tipo de capa de presentación).

La filosofía con la que se construye una aplicación monolítica es muy diferente a la de una aplicación de microservicios. Podemos pensar que construir una aplicación monolítica es como construir un rascacielos, pero en cambio una aplicación de microservicios es como construir un barrio, lo que conlleva que necesitemos construir la infraestructura (agua, luz, carreteras...) y tener en cuenta el territorio que rodea a la zona. Esta analogía nos ayuda a entender que no sólo es importante construir los componentes (edificios) en una arquitectura de microservicios, sino que además debemos tener en cuenta la infraestructura que los rodea, ya que los microservicios se ejecutan en un ecosistema donde el intercambio de información es la clave de su funcionamiento.

Una vez entendida la diferencia conceptual entre las dos arquitecturas, podemos pasar a definir cómo sería el proceso de transición de un monolito a una arquitectura basada en microservicios. Para abordar esta tarea, tendremos que realizar los siguientes pasos:

- Desgranar módulos: La tarea más difícil en todo el proceso es la primera que llevaremos a cabo, que será identificar y desgranar los componentes de la aplicación en servicios aislados e independientes. Esta tarea es esencial ya que fallar en ella requerirá revisar componentes, lo cual en principio puede no ser muy problemático, pero el verdadero problema viene con los componentes ya construidos en los que por problemas con la definición de requisitos tenemos que empezar de cero y eliminarlos.

- Tecnología: Para realizar la transición debemos tener en cuenta los diferentes frameworks que podemos elegir y valorar sus diferentes características en función de nuestras necesidades, para poder realizar esta transición correctamente.

- Estructura del equipo: Debemos tener en cuenta que la mejor manera de desarrollar microservicios es de manera independiente. Esto se debe reflejar en nuestro equipo, ya que repartir las tareas adecuadamente nos garantiza un desarrollo adecuado de los mismos.

- Base de datos: Prácticamente al mismo nivel de importancia que desgranar los módulos de la aplicación tenemos la tarea de dividir la base de datos. Esto es porque debemos usar el mismo procedimiento que con los módulos, dividiendo la base de datos paso a paso en estructuras más pequeñas. El punto más delicado de este apartado será eliminar las claves foráneas de las tablas, ya que los microservicios deben funcionar de manera independiente del resto.

- Integridad: Al separar los módulos en microservicios independientes, debemos asegurarnos de que las transacciones entre los mismos mantienen la integridad de los datos.

El despliegue de microservicios y la nube

El despliegue es el momento más arriesgado en todo el ciclo de vida de un sistema software. Una analogía que nos puede ayudar a entender la crítico de este proceso es como si tuviésemos que cambiar la rueda de un coche (hasta aquí todo normal), lo que ocurre es que el coche está moviéndose a 100km/h, lo cual vuelve la tarea extremadamente peligrosa. Aunque este sea un ejemplo extremo, los microservicios ayudan a mitigar el riesgo de esta tarea. Aún así, los principales retos a los que nos enfrentamos en el despliegue de un microservicio son:

- Mantener la estabilidad del sistema cuando nos enfrentamos a un gran volumen de actualizaciones y adiciones de componentes.

- Evitar acoplamientos fuertes entre componentes que acaben afectando a los procesos de build o despliegue por dependencia entre ambos.

- Publicar cambios muy novedosos a la API de un servicio, lo cual puede afectar negativamente a los clientes de este.

- Retirar servicios.

Estos riesgos se mitigan con las características que nos aporta la arquitectura de microservicios como las tareas de monitorización, ensamblaje del log, balanceadores de carga y la tubería de despliegue de la CI/CD. Como consejo para reducir aún más el riesgo de este momento podemos recomendar publicar cambios pequeños y poder predecir mejor qué podría fallar.

A la hora de construir nuestra aplicación, debemos elegir entre diferentes maneras de desplegar los microservicios:

- Servidor físico: En la práctica esta solución es la menos utilizada, ya que los servidores son muy costosos y aumentar su capacidad no es un proceso rápido puesto que necesitamos disponer de más servidores.

- Imagen de una máquina virtual: Es la tecnología principal que utilizan los proveedores de servicios en la nube. Es una solución muy rápida tanto para desplegar como apagar instancias de microservicios en eventos como aumento de la demanda o fallos de un servicio.

- Contenedores: Son una opción que ha aumentado en popularidad, dado que ofrecen una flexibilidad muy grande. Estos se despliegan en instancias de un sistema operativo que están alojadas en contenedores. Son más ligeros que las imágenes de las máquinas virtuales, aunque requieren conocimiento avanzado para gestionarlos.

Para asegurar el buen funcionamiento de nuestra aplicación, es de vital importancia que las instancias de los servicios se puedan iniciar con rapidez y que las diferentes instancias de un mismo servicio sean indistinguibles.

Es en este punto donde entran en juego los servicios en la nube y sus proveedores. Actualmente son soluciones muy populares que nos brindan, entre otras muchas cosas, la capacidad de escalar de manera horizontal los servicios que sufren una alta carga (creando más instancias de ese mismo servicio), sobrellevando ese estrés de una manera muy elegante y sin necesidad de disponer de los recursos físicos necesarios, únicamente se nos cobrará por el servicio que estamos recibiendo.

Las opciones más recomendables y eficientes para crear una aplicación basada en microservicios son las de servicios en la nube (imágenes virtuales y contenedores), porque nos ofrecen una gran flexibilidad y elasticidad para gestionar rápidamente nuestros microservicios, gran capacidad para escalar horizontalmente y contener errores en nuestra aplicación.

Riesgos y dificultades que presentan

Hasta ahora hemos visto las bondades y todo el potencial de la arquitectura de microservicios. Pero no es oro todo lo que reluce, y esta arquitectura viene acompañada de una serie de problemas intrínsecos, además de los del desarrollo de la lógica del negocio, que debemos tener muy en cuenta antes de optar por desarrollar una aplicación con esta arquitectura. Podemos enumerar los siguientes riesgos y dificultades que tendremos que valorar:

- Identificar y dividir los diferentes microservicios requiere un gran conocimiento sobre el dominio del problema. Puede que necesitemos refactorizar diferentes fuentes de código o migrar datos de una base de datos a otra si nuestra tarea es hacer una transición de monolítico a microservicios. Todo este proceso si no es tratado con sumo cuidado y con el conocimiento suficiente, puede llevarnos a que no hayamos identificado correctamente las dependencias, lo cual puede generar incompatibilidades o errores en el despliegue.

- Al ser un sistema distribuido nuestra única preocupación no es construir únicamente el código de la aplicación, sino que tenemos que crear la infraestructura de CI/CD para automatizar y monitorizar la aplicación. Esto requiere de una gran experiencia y conocimiento en DevOps, ya que fallar en esta infraestructura nos impide avanzar en el desarrollo de la aplicación.

- Encontrar los fallos entre la multitud de servicios que compone la aplicación es muy complejo ya que existen fuentes de fallos adicionales a los habituales en una arquitectura monolítica, debido a la heterogeneidad y comunicación de los servicios.

- También debemos tener en cuenta que podemos incurrir en las siguientes falacias como suponer que la red es fiable, la latencia es nula, el ancho de banda es infinito o que el coste de transporte es nulo. Debemos tener en cuenta cómo puede ser la ejecución de un servicio en un entorno inestable y cómo volver hacia atrás cuando hay errores en las transacciones.

- En el caso de que un servicio falle o no se encuentre disponible, debemos establecer mecanismos de acción que solucionen, o si no es posible, contengan este fallo para que no se propague a diferentes partes de la aplicación.

Tras analizar los diferentes riesgos, debemos valorar si estamos dispuestos a abordar las dificultades que acompañan a esta arquitectura o si tenemos el conocimiento suficiente para ello.

Cuando no usar microservicios

Aunque sean una arquitectura muy popular y parece estar en boca de todos los desarrolladores, los microservicios no son una solución universal y en este apartado vamos a explorar cuáles son los escenarios en los que una arquitectura de microservicios no sería deseable:

- Traen consigo una gran complejidad que las aplicaciones monolíticas no poseen a la hora de desarrollarlas. Definir los requisitos de cada servicio es una tarea que conlleva un gran esfuerzo. Si por motivos económicos o temporales no se desea invertir en todas las tareas que acompañan a los microservicios tales como la monitorización o la escalabilidad, no debemos optar por esta arquitectura.

- Una aplicación basada en microservicios puede tener una gran cantidad de servidores que usa simultáneamente, en la que se ejecutan instancias de los servicios. Aunque la nube nos facilita la tarea de gestión de los servidores y abarata costes, si no se está preparado para gestionar y monitorizar la complejidad operacional de tantos servidores, esta arquitectura podría ser un problema más que una solución.

- La arquitectura de microservicios está orientada hacia la escalabilidad y la reusabilidad. Si nuestro objetivo es diseñar una aplicación pequeña y con una base de usuarios también pequeña, sería un error optar por esta arquitectura dados los costes que conlleva adicionales al desarrollo del código fuente.

- Si la aplicación que queremos desarrollar maneja multitud de datos diferentes y requieren muchas transformaciones complejas y agregaciones, los microservicios tampoco son una buena idea. Esto es por su naturaleza distribuida, sería muy complicado definir las responsabilidades de cada microservicio y se solaparían unos con otros, pudiendo violar el principio de responsabilidad única.

Spring Boot

Spring Boot es uno de los 21 proyectos que tiene activos el equipo de Spring, así que para poder explicar y entender este framework debemos abordar primero el corazón de todos estos proyectos, el cuál es Spring Framework.

Spring Framework está construido alrededor de la idea de la Inyección de Dependencias (DI), la cual vamos a explicar de manera detallada.

La DI es un patrón que implementa el principio de Inversion of Control (IoC), el cual se basa en que el control de los objetos se delega a un contenedor o un framework. En contraste con la programación tradicional, en la que nuestro código realiza llamadas a una librería, IoC permite al framework tomar el control del flujo de un programa y hacer llamadas a nuestro código. Para conseguir esto los diferentes frameworks usan abstracciones con comportamientos definidos, los cuáles podemos modificar extendiendo estas clases o construyendo las nuestras propias. Este principio nos ofrece las siguientes ventajas:

- Desacoplar la ejecución de una tarea de su implementación.
- Facilitar el cambio de las implementaciones.
- Incrementar la modularidad del programa.
- Facilita el testeo de un programa porque nos permite aislar componentes o simular sus dependencias.

Existen diferentes mecanismos para conseguir la IoC como Factory Pattern o Strategy Design Pattern, pero queremos analizar la que utiliza Spring, que es la DI.

Normalmente, crearemos un objeto de la siguiente manera:

```
public class Store{
  private Item item;
  public Store() {
    item= new ItemImpl1();
  }
}
```

Como podemos observar, necesitamos instanciar una implementación de la interfaz Item dentro de la clase Store(). En cambio, si usamos DI podemos reescribir la clase sin necesidad de especificar la implementación de Item que usaremos.

```
public class Store{
  private Item item;
  public Store(Item item) {
    this.item=item;
  }
}
```

El contenedor de Spring es el encargado de crear los objetos, configurarlos, conectarlos y gestionarlos durante su ciclo de vida completo, desde que son creados hasta que son eliminados.

El contenedor de Spring está representado en el ApplicationContext. Este es quien se encarga de controlar todas las clases y gestionarlas de manera correcta, proporcionando las dependencias necesarias (teniendo en cuenta que las hayamos declarado correctamente).

```
ApplicationContext ctx = new
AnnotationConfigApplicationContext(someConfigClass);
Ctx.getBean(Class.class)
```

Como podemos ver en el ejemplo necesitamos una clase para la configuración del Application Context. Esta clase debería ser algo parecido a esto:

```
@Configuration
public class MyApplicationContextConfiguration {

    @Bean
    public DataSource dataSource() {
        MysqlDataSource dataSource = new MysqlDataSource();
        dataSource.setUser("root");
        dataSource.setPassword("s3cr3t");
        dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
        return dataSource;
    }

    @Bean
    public UserDao userDao() {
        return new UserDao(dataSource());
    }
}
```

Esta clase se interpretará como la configuración del ApplicationContext gracias a la anotación @Configuration, mientras que los objetos están anotados con @Bean (que explicaremos en breve). Además de esta manera de construir el ApplicationContext, también se puede hacer a través de un XML o clases de Java anotadas, pero en este caso la interfaz ApplicationContext es la más clara para entender el funcionamiento y objetivo de esta.

La anotación Bean que vemos en las dependencias que vamos a crear indica que los objetos que sean creados por ellas serán Beans. La implicación directa de este "marcador" es que Spring reconocerá estas instancias como propias, y se responsabilizará de su gestión. Si queremos especificar cuántas instancias debe generar Spring, podemos conseguirlo con la anotación @Scope. Podemos usar @Scope("singleton") para crear un singleton, es decir, una instancia única. Esta configuración es la más habitual, peor también existen otras opciones como @Scope("prototype"), que creará una instancia nueva cada vez que alguien referencie a ese Bean, o @Scope("session"), que crea una instancia nueva para cada sesión HTTP.

Podemos ahorrarnos tener que realizar las llamadas a las nuevas instancias usando la anotación @ComponentScan(). Esta anotación, que acompaña a la clase encargada del ApplicationContext, escanea el paquete en el que se encuentra y busca Beans que puedan ser inyectadas. Spring reconocerá las Beans gracias a la anotación @Component(), que debemos usarla para etiquetar las Beans y que Spring pueda reconocerlas. Sólo nos queda una parte por resolver en este esquema después de que Spring ya sepa cómo reconocer las Beans, y es concretar dónde debe inyectarlas. Con la anotación @Autowired, que la usaremos al declarar las instancias en el ApplicationContext, Spring sabrá que debe inyectar esa dependencia cuando sea necesario, es decir, cuando alguna clase la use como parámetro de su constructor. Con el paso del tiempo Spring tiene mecanismos para reconocer las dependencias hasta cuando no están anotadas con @Autowired, pero su uso es altamente recomendable.

Tras esta explicación de etiquetas, vamos a concretar toda esta teoría repasando las diferentes maneras en las que podemos aplicar el patrón de DI con Spring.

- DI mediante constructor:

```
private DataSource dataSource;

private UserDao(@Autowired DataSource dataSource) {
    this.dataSource = dataSource;
}
```

- DI mediante campo:

```
@Autowired
private DataSource dataSource;
```

- DI mediante setter:

```
private DataSource dataSource;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
```

Con los tres métodos conseguimos el mismo resultado, nuestro Bean será inyectado y funcionará correctamente. Las diferencias entre los métodos y cuál es la mejor son objeto de debate, y hoy en día no hay una manera que sea siempre la más efectiva. La recomendación oficial de Spring es que, para las dependencias obligatorias, se use la DI mediante constructor, y para las opcionales mediante campo/setter.

Spring ofrece muchas otras características de una gran importancia y que le han llevado a ser tan importante como lo es actualmente, pero nuestro objetivo es conocer cuál es el funcionamiento de Spring Boot, y a continuación vamos a detallar cuáles son las características más notables que nos ofrece este proyecto de Spring.

Como ya sabemos, Spring Boot está construido sobre Spring Framework. En pocas palabras, el objetivo con el que surge Spring Boot es ayudarnos a crear, configurar y ejecutar tanto aplicaciones sencillas como orientadas a web de una rápida y sencilla.

A diferencia de Spring Framework, donde se necesita configurar todo a mano, Spring Boot nos salva de este problema. Spring Boot se encargará de elegir las dependencias, autoconfigurar todas las características que vamos a usar y nos permite ejecutar nuestra aplicación con un solo click. Además, nos facilita el proceso de despliegue de nuestra aplicación.

Puede parecer "mágico" todo lo que Spring Boot puede hacer por nosotros, pero detrás de esa apariencia hay un framework que está diseñado para realizar estas tareas con una aparente sencillez que es digna de asombro. Vamos a enumerar y ejemplificar las características más importantes que convierten a Spring Boot en uno de los frameworks más cómodos para el desarrollo de código:

- Autoconfiguración "inteligente": Consideremos que hemos añadido una dependencia a nuestro pom.xml (que contiene la información del proyecto y su configuración para el proceso de build) que, por ejemplo, relacionada con una base de datos. En este caso, Spring Boot asume que probablemente, uno de los objetivos de la aplicación sea establecer conexión con la base de datos, entonces prepara la aplicación para el acceso a bases de datos. Además, si es una base de datos específica, preparará el acceso específico para esa base de datos. Todo esto se puede realizar simplemente añadiendo @EnableAutoConfiguration a nuestro proyecto.
- Aplicación única: Para ejecutar una aplicación web basada en Java necesitamos empaquetar nuestra aplicación, elegir qué tipo de servidor web vamos a usar, configurarlo y descargarlo, y tras todo esto debemos organizar el proceso de despliegue. Con Spring Boot únicamente tenemos que empaquetar y ejecutar la aplicación con un único click o comando, gracias que posee un servidor web Tomcat embebido y lo usa para desplegar nuestra aplicación.
- Pragmático: Spring Boot tiene un enfoque pragmático a la hora de construir una aplicación, teniendo como prioridad facilitar que los desarrolladores puedan centrarse en su tarea principal, el desarrollo. Por ese motivo, nos ofrece Spring Initializr, el cual nos ayuda a crear el esqueleto de nuestra aplicación (En qué lenguaje está escrito, qué versiones utiliza, la estructura del proyecto, que dependencias posee... etc.) de una manera súper sencilla y rápida.

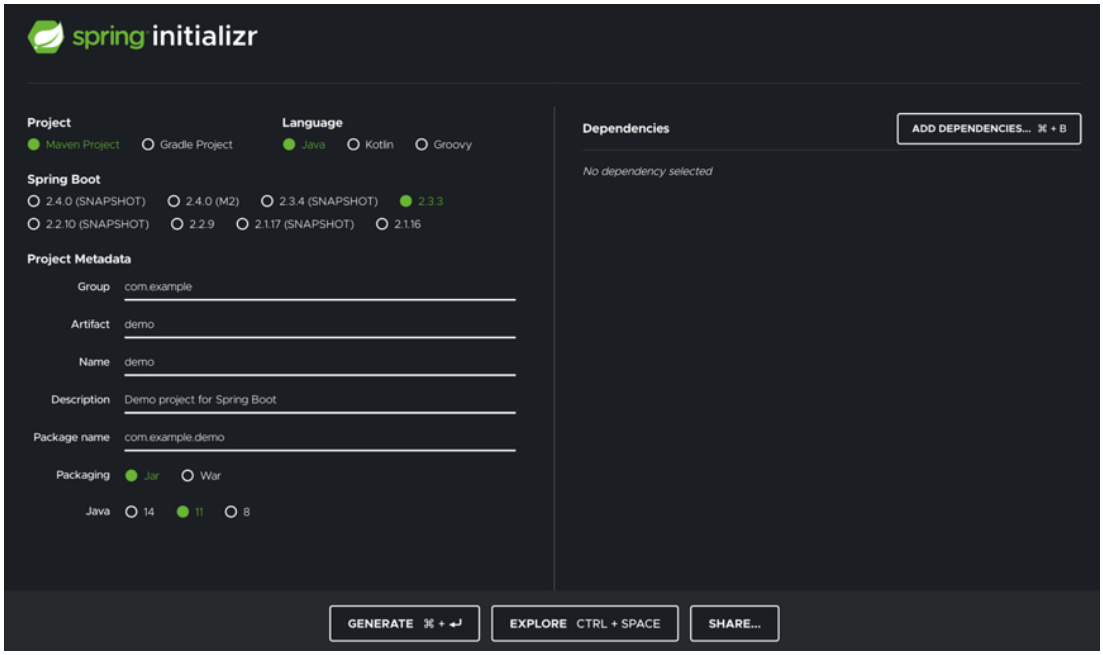

Ilustración 1: Spring Initializr

En este apartado hemos revisado las características principales de Spring Boot, que veremos reflejadas en la aplicación y se detallará cómo funcionan en el contexto de esta. En el siguiente apartado revisaremos las tecnologías que hemos considerado para desarrollar la aplicación y que serán vitales para el funcionamiento de una aplicación basada en la arquitectura de microservicios.

Spring Cloud Netflix:

En una aplicación basada en la arquitectura de microservicios queremos que los servicios tengan una comunicación fluida. Aquí es donde entra Spring Cloud Netflix, que nos ayudará a implementar ciertas características que pudimos ver en el apartado de patrones. Entre ellos, podemos destacar:

- Localización de Servicio
- Enrutamiento de Servicio
- Balance de carga
- Patrón de cortocircuito

Existen diferentes tecnologías que nos ofrecen estas características dentro de Spring Cloud Netflix, siendo las más importantes Eureka y Zuul.

Eureka es un servicio REST que actúa como un servidor de descubrimiento, que localiza y registra los microservicios desplegados (que estén configurados para ello), para poder tener constancia de su salud y características generales. Su funcionamiento es tan simple como que cuando tenemos levantado nuestro servidor, los servicios clientes que estén configurados para conectarse con Eureka, lo buscarán y Eureka les notificará que han sido registrados, manteniendo una comunicación constante a través de “latidos” cada 30 segundos. De esta manera Eureka tiene constancia del estado de sus servicios

clientes, pudiendo eliminar de su registro a los microservicios “muertos” (que no hayan latido pasados 30 segundos). Además, cada servicio conectado al servidor puede recuperar el registro de los servicios conectados para tener constancia del ecosistema completo. Eureka puede funcionar a modo de clúster si lo configuramos de tal manera, para que existan varias instancias del servidor. En resumen, Eureka nos proporciona enormes ventajas en una aplicación basada en microservicios, pudiendo abstraer las direcciones físicas de los servicios y ayudarnos a tener constancia del estado del ecosistema de la aplicación.

Más adelante se detallará cómo implementar este servidor y cómo configurar los servicios para que puedan ser registrados.

Zuul es un edge service (o servicio frontera) que nos ayuda a aplicar el patrón API Gateway. Esto significa que nos permite filtrar y enrutar todas las peticiones a nuestra aplicación dinámicamente. Aunque aparentemente es un servicio más dentro del ecosistema de la aplicación, se comunica con Eureka para solicitar las instancias de los servicios desplegados y realiza las peticiones a estos. A primera vista, puede parecer una desventaja que la aplicación pueda tener aparentemente un único punto de fallo, pero al poder replicar las instancias de este servicio sobrepasamos este problema sin mayor dificultad. Además, nos ofrece la capacidad de filtrar las peticiones según los filtros que definamos y balancea automáticamente la carga de su tráfico. Al configurar Zuul, automáticamente dispondremos de Hystrix y Ribbon en nuestra aplicación ya que son una parte intrínseca de Zuul. Ambas son dos herramientas que también pertenecen a Spring Cloud Netflix y aportan características adicionales a nuestro servicio de frontera. Hystrix proporciona tolerancia a fallos y resiliencia gracias a que implementa el patrón de cortocircuito (circuit-breaker). Ribbon por su parte nos permite el balanceo de carga de peticiones. Además, podemos añadir Spring Security para securizar el acceso a las rutas que intercepta este servicio, lo cuál es muy útil para implementar un sistema de inicio de sesión, por ejemplo.

Al igual que Eureka, más adelante se detallará cómo implementar y configurar este servicio frontera.

Spring Security

La seguridad es un tema central en cualquier aplicación que se desarrolle hoy en día. Spring Security nos ofrece soluciones sencillas para securizar nuestras aplicaciones sean tanto de microservicios como no. Nos interesa securizar nuestra aplicación ya que no queremos que se cualquiera se pueda comunicar con ellos y hacer las peticiones que quiera. De esto se encargará Zuul, que, en conjunto con el servicio de autenticación, filtrará las peticiones. A continuación, vamos a describir las bases teóricas de Spring Security para poder implementarlo después en nuestra aplicación.

Spring Security, en pocas palabras, es una serie de filtros que podemos aplicar al servidor embebido Tomcat de una aplicación Spring Boot. Estos filtros, entre otras cosas, nos

permiten implementar autenticación y autorización a nuestra aplicación. La autenticación es poder verificar que un usuario es realmente quien dice ser, y la autorización es definir si un usuario autenticado puede realizar una tarea o no dependiendo del rol que tenga asignado en la misma. Estos filtros se "sitúan" antes de la llamada al controller, lo que nos permite definir quién puede acceder a ese recurso y quién no.

Spring Security nos ofrece varios filtros que extender, pero sólo nos incumben dos, que son UsernamePasswordAuthenticationFilter, que nos ofrece la posibilidad de identificar una petición post, con nombre de usuario y contraseña e intenta autenticarla, y OncePerRequestFilter, que nos permitirá filtrar todas las llamadas como queramos.

Para configurar Spring Security necesitamos crear un WebSecurityConfigurerAdapter, que filtrará las peticiones por criterios como origen, si está autenticada la petición o la url a la que se quiera acceder. También necesitaremos un último filtro, que será UsernamePasswordAuthenticationFilter, que, como su nombre indica, nos permite filtrar según el nombre de usuario y la contraseña introducida. Como última característica necesaria para nuestro proyecto, estará el UserDetailsService, que es necesario implementarlo en una clase para que podamos manejar los usuarios de la aplicación.

Servicios REST con Spring Boot

La parte más importante del funcionamiento de los microservicios y de todo tipo de aplicaciones web es la comunicación. Hoy en día, los servicios REST son el estándar de facto para crear API's que se comuniquen a través de Internet.

Los servicios REST (Representational State Transfer), exponen una serie de recursos (a través de Internet) a los que se puede acceder con peticiones HTTP sencillas del tipo GET, PUT, POST o DELETE. Existen más peticiones, pero las más comunes son las cuatro mencionadas. Estas peticiones nos permiten acceder a recursos que consisten en datos que se ofrecen, o poder operar sobre ellos. Cada recurso se identifica de diferentes maneras: URL, método HTTP, parámetros, tipo de respuesta o tipo consumido. Estas peticiones pueden ir acompañadas de datos adicionales a la propia petición o respuesta, que pueden ser de tipos muy variados, desde Text o XML hasta JSON.

En Spring, el módulo web-MVC se encarga de proporcionar el soporte necesario para crear servicios REST. Como casi todos los módulos, incluye sus propias anotaciones específicas. Para implementar este módulo, debemos anotar una clase con la etiqueta @RestController. Además, los métodos HTTP tienen asociados una etiqueta específica, como @GetMapping, @PostMapping, ... etc.

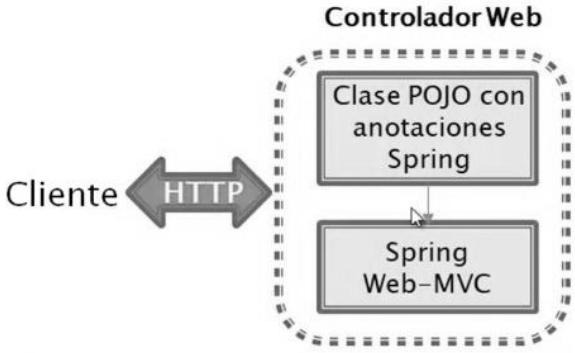

Ilustración 2: REST y Spring Boot

Microservicios en Spring Boot

Un microservicio en Spring Boot es muy similar a una aplicación normal que podemos desarrollar. La diferencia radica en que estará preparado para enviar y recibir datos de otros microservicios de la aplicación y trabajar en conjunto con ellos, siendo una pieza de un esquema mucho más grande. La estructura estándar de un microservicio es la siguiente:

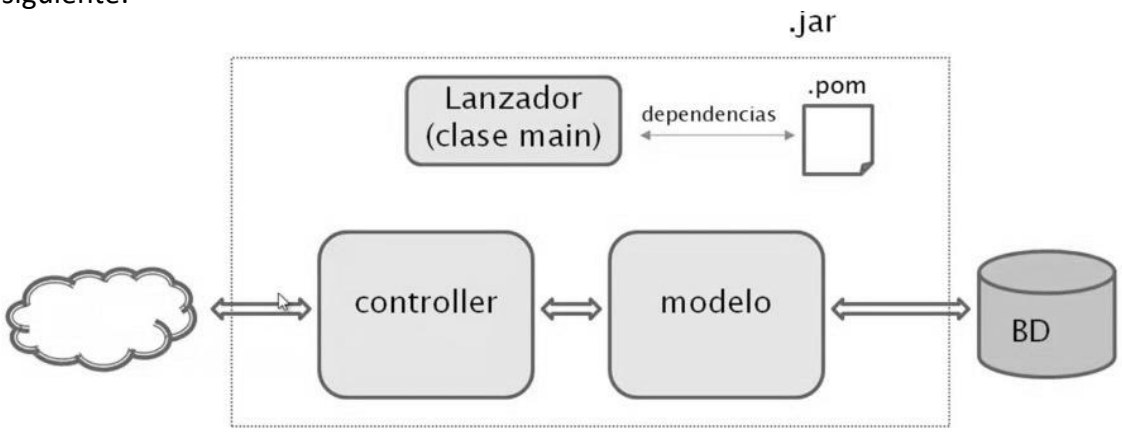

Ilustración 3: Estructura estándar microservicio

Las partes que observamos en la estructura serán fáciles de identificar si estamos familiarizados con las API's web. La diferencia más notable en Spring Boot es que la clase main generalmente será algo parecido a esto:

```
@SpringBootApplication
public class Application {
        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);
        }
}
```

Ilustración 4: @SpringBootApplication

Habitualmente el main suele realizar llamadas a otros métodos o clases de la aplicación, pero gracias a Spring Boot y sus Beans, no necesitamos prácticamente codificar la clase main. La anotación @SpringBootApplication es vital para el proyecto, ya que resume tres anotaciones en ella, que son @Configuration, @EnableAutoConfiguration y @ComponentScan. Estas anotaciones nos permiten, respectivamente, definir los parámetros de configuración del servicio, dejar que Spring Boot configure automáticamente nuestra aplicación en función de las dependencias del proyecto y, por último, escanear el paquete para buscar Beans que puedan ser inyectadas. Todas estas características deberían sernos familiares, ya que poco a poco vamos introduciendo ejemplos prácticos de los conceptos que hemos definido anteriormente de Spring Boot. En el diseño de la solución explicaremos más conceptos que todavía no hemos puesto en práctica y sus implementaciones específicas

Una parte muy importante y que necesitamos entender para el funcionamiento de la aplicación es el ciclo de vida de los servicios en Spring Boot. En primer lugar, este debe ser empaquetado y desplegado de manera independiente. En Spring Boot esto se realiza usando Maven (aunque existen alternativas como Gradle) para crear el ejecutable del microservicio y ponerlo en marcha. En sus primeros instantes de vida, el servicio entra en su segunda fase, que es la de bootstrap, lo que significa que cargará los datos de configuración que necesite para empezar a funcionar. Como tercer paso el microservicio antes de poder recibir llamadas para ejecutar sus responsabilidades necesita comunicarse con un agente localizador de servicios, que registrará su IP (única y no permanente) y gestionará las solicitudes para consumir ese servicio. Tras superar esta fase, nuestro microservicio está funcionando y la única tarea que tenemos que llevar a cabo es la monitorización. Esta tarea la lleva a cabo el localizador de servicios, asegurando que el microservicio mantiene una buena “salud” durante su ejecución. Si éste descubre algún problema con el servicio, se encargará de bien apagarlo o levantar más instancias de este para superar momentos de gran estrés debidos a un aumento en la demanda.

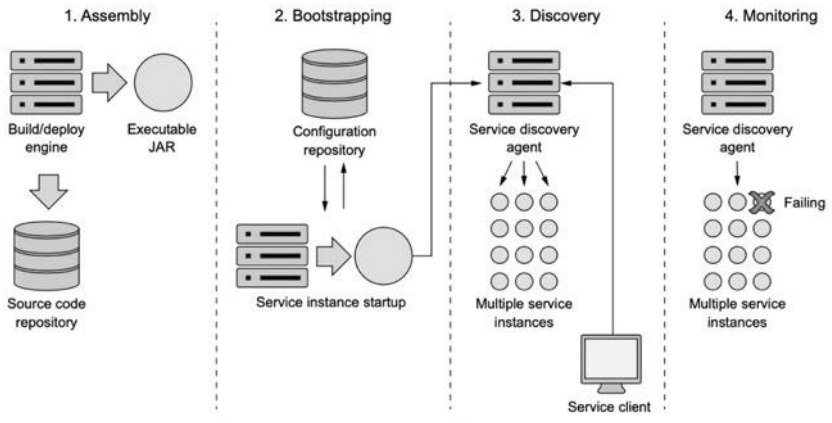

Ilustración 5: Ciclo de vida de un microservicio

Análisis del problema

Captura y definición de requisitos

En este apartado vamos a reflexionar y definir los aspectos que darán forma a nuestra aplicación. El método para definir los requisitos que vamos a usar son las Historias de Usuario (HdU), un método muy popular entre las metodologías ágiles, que nos permite definir características de la aplicación de manera general e informal, vista desde la perspectiva del usuario. Esta manera de definir las metas de nuestro proyecto nos permite encontrar soluciones creativas y tener un objetivo claro en el desarrollo.

Como características principales que una HdU debe cumplir para que esté bien definida y realmente nos ayude a aportar valor al cliente, podemos mencionar las siguientes:

- Independientes: En la medida de lo posible, su función debe ser atómica y no depender de otras HdU.
- Negociables: Deben ser lo suficientemente ambiguas para permitir un desarrollo creativo y dejando su concreción para los criterios de aceptación.
- Valoradas: El cliente debe valorar las metas positivamente.
- Estimables: Aunque la estimación sea una tarea complicada, en la medida de lo posible deberíamos definir su posible alcance.
- Pequeñas: De manera recomendable, estas deben tener una duración mayor de un día y menor a dos semanas.
- Verificables: Se puede palpar cuando se ha cumplido la meta gracias a los criterios de aceptación.

Los criterios de aceptación son simplemente las pruebas que debe pasar la meta de una HdU para poder considerar que está completada.

Tras este repaso al concepto y justificación teórica de las HdU, vamos a definirlas teniendo en cuenta el planteamiento del problema:

Título: Inicio de sesión	Prioridad: Muy Alta	Estimado: 20 horas
Como usuario quiero iniciar sesión en la aplicación para poder usar sus funcionalidades		
Criterios de aceptación: • Cuando inicie la aplicación, se debe mostrar esta ventana de bienvenida. • En la ventana de bienvenida aparecerán dos cuadros de texto, uno será el nombre de usuario y otro la contraseña. • También debe aparecer un botón para confirmar el inicio de sesión. • No se avanzará a la aplicación hasta que no se consiga un inicio de sesión satisfactorio.		

Ilustración 6: Inicio de sesión HdU

Título: Página principal top películas	Prioridad: Media	Estimado: 1 hora
Como usuario quiero una pagina principal que contenga las películas con mejor nota del catálogo para poder elegir entre las películas mejor valoradas		
Criterios de aceptación: • Que cuando acceda a la página principal de la aplicación se carguen las 10 películas con mejor nota. • Que muestre correctamente las películas con mejor valoración que se encuentran en la BBDD de películas. • Que, al poner el ratón sobre una película, su portada se resalte para diferenciarse del resto. • Que, al clicar una película, la aplicación nos redirija a los detalles de esa película correctamente. • Que sean diez películas las que se muestren en la lista. • Que no se solape con el resto de las listas.		

Ilustración 7: Top películas HdU

Título: Página principal top series	Prioridad: Media	Estimado: 1 hora
Como usuario quiero una pagina principal que contenga las series con mejor nota del catálogo para poder elegir entre las series mejor valoradas		
Criterios de aceptación: • Que cuando acceda a la página principal de la aplicación se carguen las 10 series con mejor nota. • Que muestre correctamente las series con mejor valoración que se encuentran en la BBDD de series. • Que, al poner el ratón sobre una serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una serie, la aplicación nos redirija a los detalles de esa serie correctamente. • Que sean diez series las que se muestren en la lista. • Que no se solape con el resto de las listas.		

Ilustración 8: Top series HdU

Título: Página principal películas	Prioridad: Alta	Estimado: 1 hora
Como usuario quiero una pagina principal que contenga todas las películas para poder elegir en función de mis preferencias personales		
Criterios de aceptación: • Que cuando acceda a la página principal de la aplicación se carguen todas las películas • Que muestre correctamente las películas que se encuentran en las BBDD de películas. • Que, al poner el ratón sobre una película o serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una película o serie, la aplicación nos redirija a los detalles de esa película o serie correctamente. • Que no se solape con el resto de las listas.		

Ilustración 9: Home Films HdU

Título: Página principal series	Prioridad: Alta	Estimado: 1 hora
Como usuario quiero una pagina principal que contenga todas las series para poder elegir en función de mis preferencias personales		
Criterios de aceptación: • Que cuando acceda a la página principal de la aplicación se carguen todas las series • Que muestre correctamente las series que se encuentran en las BBDD de series. • Que, al poner el ratón sobre una serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una serie, la aplicación nos redirija a los detalles de esa serie correctamente. • Que no se solape con el resto de las listas.		

Ilustración 10: Home series HdU

Título: Barra de navegación	Prioridad: Alta	Estimado: 3 horas
Como usuario quiero una barra de navegación en la parte superior para poder elegir entre inicio, series, películas o búsqueda		
Criterios de aceptación: • Que en cualquier parte de la aplicación se muestre la barra de navegación. • Que la barra de navegación contenga los botones: Home, Films, Series y Search • Que la barra de navegación nos lleve correctamente a las diferentes partes de la aplicación. • Que la barra esté fija en la parte superior independientemente del desplazamiento vertical. • Que la barra sea transparente para no tapar los contenidos, por motivo estético y práctico.		

Ilustración 11: Navbar HdU

Título: Películas catálogo	Prioridad: Muy Alta	Estimado: 15 horas
Como usuario quiero ver todo el catálogo de películas en un apartado de películas para poder observar las que ofrece la aplicación.		
Criterios de aceptación: • Que cuando acceda al apartado Films de la aplicación se muestre una lista que contenga todas las películas. • Que las portadas se visualicen correctamente. • Que muestre correctamente las películas que se encuentran en la BBDD de películas. • Que, al poner el ratón sobre una película, su portada se resalte para diferenciarse del resto. • Que, al clicar una película, la aplicación nos redirija a los detalles de esa película correctamente. • Que no se solape con el resto de las listas.		

Ilustración 12: Catálogo Films HdU

Título: Películas géneros	Prioridad: Alta	Estimado: 2 horas
Como usuario quiero ver las películas del catálogo clasificadas por géneros en un apartado de películas para poder ver el catálogo de películas según mis preferencias personales		
Criterios de aceptación: • Que cuando acceda al apartado Films de la aplicación se muestren varias listas de películas que sean de los géneros: acción, drama, aventuras, crimen y misterio. • Que las portadas se visualicen correctamente. • Que muestre correctamente las películas con los géneros indicados que se encuentran en la BBDD de películas. • Que, al poner el ratón sobre una película, su portada se resalte para diferenciarse del resto. • Que, al clicar una película, la aplicación nos redirija a los detalles de esa película correctamente. • Que no se solapen con el resto de las listas.		

Ilustración 13: Géneros Films HdU

Título: Películas características	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada película tenga unas características generales como duración, género o director para poder elegirlas según mis preferencias personales.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una película se muestren sus detalles, entre ellos sus características. • Que muestre correctamente las características de la película que se encuentra en la BBDD de películas. • Que las características que muestren sean: Duración en minutos, género/s, director, año de estreno y valoración de la crítica en base 10. • Que no se solape con otros elementos de los detalles de la película.		

Ilustración 14: Características Films HdU

Título: Películas sinopsis	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada película tenga una sinopsis para poder conocer brevemente la trama de la película.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una película se muestren sus detalles, entre ellos su sinopsis. • Que muestre correctamente la sinopsis de la película que se encuentra en la BBDD de películas. • Que no se solape con otros elementos de los detalles de la película.		

Ilustración 15: Sinopsis Films HdU

Título: Películas portada	Prioridad: Alta	Estimado: 1 hora
Como usuario quiero que cada película tenga una portada para poder identificarla de un vistazo rápido		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una película se muestren sus detalles, entre ellos su portada. • Que muestre correctamente la portada de la película que se encuentra en la BBDD de películas. • Que no se solape con otros elementos de los detalles de la película.		

Ilustración 16: Portada Films HdU

Título: Películas tráiler	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada película tenga un enlace a su tráiler para poder tener un avance audiovisual de la película.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una película se muestren sus detalles, entre ellos el enlace a su tráiler. • Que el enlace al tráiler de la película sea el que se encuentra en la BBDD de películas. • Que no se solape con otros elementos de los detalles de la película.		

Ilustración 17: Trailer Films HdU

Título: Películas comentarios	Prioridad: Muy Alta	Estimado: 5 horas
Como usuario quiero que cada película tenga una sección de comentarios para poder ver las opiniones de los demás usuarios.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una película se muestren sus detalles, entre ellos sus comentarios. • Que los comentarios de la película sean los que se encuentra en la BBDD de reviews y se correspondan con la película. • Que no se solape con otros elementos de los detalles de la película. • Que los comentarios contengan nombre de usuario, nota y una pequeña opinión. • Que los usuarios puedan comentar poniendo su propia opinión y nota.		

Ilustración 18: Comentarios Films HdU

Título: Series catálogo	Prioridad: Muy Alta	Estimado: 15 horas
Como usuario quiero ver todo el catálogo de series en un apartado de series para poder observar las que ofrece la aplicación.		
Criterios de aceptación: • Que cuando acceda al apartado Series de la aplicación se muestre una lista que contenga todas las series. • Que las portadas se visualicen correctamente. • Que muestre correctamente las películas que se encuentran en la BBDD de series. • Que, al poner el ratón sobre una serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una serie, la aplicación nos redirija a los detalles de esa serie correctamente. • Que no se solape con el resto de las listas.		

Ilustración 19: Catálogo Series HdU

Título: Series géneros	Prioridad: Alta	Estimado: 2 horas
Como usuario quiero ver las series del catálogo clasificadas por géneros en un apartado de series para poder ver el catálogo de series según mis preferencias personales.		
Criterios de aceptación: • Que cuando acceda al apartado Series de la aplicación se muestren varias listas de series que sean de los géneros: drama, acción, crimen, ciencia ficción y comedia. • Que las portadas se visualicen correctamente. • Que muestre correctamente las series con los géneros indicados que se encuentran en la BBDD de series. • Que, al poner el ratón sobre una serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una serie, la aplicación nos redirija a los detalles de esa serie correctamente. • Que no se solape con el resto de las listas.		

Ilustración 20: Géneros Series HdU

Título: Series características	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada serie tenga unas características generales como duración, género o temporadas para poder elegirlas según mis preferencias personales.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una serie se muestren sus detalles, entre ellos sus características. • Que muestre correctamente las características de la serie que se encuentra en la BBDD de películas. • Que las características que muestren sean: Duración en minutos de cada episodio, género/s, temporadas, episodios por temporada, año de estreno y valoración de la crítica en base 10. • Que no se solape con otros elementos de los detalles de la serie.		

Ilustración 21: Características Series HdU

Título: Series sinopsis	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada serie tenga una sinopsis para poder conocer brevemente la trama de la serie.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una serie se muestren sus detalles, entre ellos su sinopsis. • Que muestre correctamente la sinopsis de la serie que se encuentra en la BBDD de series. • Que no se solape con otros elementos de los detalles de la serie.		

Ilustración 22: Sinopsis Series HdU

Título: Series portada	Prioridad: Alta	Estimado: 1 hora
Como usuario quiero que cada serie tenga una portada para poder identificarla de un vistazo rápido.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una serie se muestren sus detalles, entre ellos su portada. • Que muestre correctamente la portada de la serie que se encuentra en la BBDD de series. • Que no se solape con otros elementos de los detalles de la serie.		

Ilustración 23: Portada Series HdU

Título: Series tráiler	Prioridad: Media	Estimado: 1 hora
Como usuario quiero que cada serie tenga un enlace a su tráiler para poder tener un avance audiovisual de la primera temporada de la serie.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una serie se muestren sus detalles, entre ellos el enlace a su tráiler. • Que el enlace al tráiler de la serie sea el que se encuentra en la BBDD de series. • Que no se solape con otros elementos de los detalles de la serie.		

Ilustración 24: Trailer Series HdU

Título: Series comentarios	Prioridad: Muy Alta	Estimado: 5 horas
Como usuario quiero que cada serie tenga una sección de comentarios para poder ver las opiniones de los demás usuarios.		
Criterios de aceptación: • Que cuando se haya clicado en la portada de una serie se muestren sus detalles, entre ellos sus comentarios. • Que los comentarios de la serie sean los que se encuentra en la BBDD de reviews y se correspondan con la serie. • Que no se solape con otros elementos de los detalles de la serie. • Que los comentarios contengan nombre de usuario, nota y una pequeña opinión. • Que los usuarios puedan comentar poniendo su propia opinión y nota.		

Ilustración 25: Comentarios Series HdU

Título: Búsqueda híbrida	Prioridad: Muy Alta	Estimado: 20 horas
Como usuario quiero poder realizar búsquedas en las que aparezcan tanto series o películas para poder encontrar el título del catálogo que busco.		
Criterios de aceptación: • Que cuando acceda al apartado Search de la aplicación se muestre una barra de búsqueda en la que introducir los criterios de búsqueda, que serán Título o género. • Que cuando se introduzcan caracteres se muestren automáticamente las películas y series que cuadren con los criterios de búsqueda. • Que las portadas se visualicen correctamente. • Que muestren correctamente las películas y las series que se encuentran en la BBDD de películas. • Que, al poner el ratón sobre una película o serie, su portada se resalte para diferenciarse del resto. • Que, al clicar una película o serie, la aplicación nos redirija a los detalles de esa película o serie correctamente. • Que no se solapen los elementos de la búsqueda.		

Ilustración 26: Búsqueda HdU

Tras repasar todas las HdU y sus criterios de aceptación, podemos tener una idea general de cómo será nuestra aplicación y las funcionalidades que nos ofrecerá. Ahora bien, necesitamos pensar también en cómo implementaremos todas estas características, por eso en el siguiente apartado trataremos de diseñar la solución que nos permita cumplir con todas las metas de las HdU.

Diseño de la solución

Este apartado tratará en detalle el proceso de desarrollo completo de nuestra aplicación, desde la decisión de las herramientas a utilizar y los lenguajes hasta el testeo de sus componentes. La aplicación tendrá como pilares principales, ya que son el objeto de estudio de este proyecto, la arquitectura basada en microservicios y el framework Spring Boot.

Análisis y justificación de las herramientas:

Eclipse

Usaremos Eclipse para desarrollar el back-end. Eclipse es un entorno de desarrollo (IDE) integrado creado por IBM y escrito principalmente en Java. Este IDE es el segundo más popular a nivel mundial, ya que principalmente está pensado para desarrollar lenguajes en el segundo lenguaje de programación más popular, Java. Además, ofrece soporte para multitud de lenguajes como Python, JavaScript o C++. Una de las características más potentes de Eclipse y uno de los motivos más importantes por lo que lo hemos elegido es porque ofrece un sistema de extensibilidad mediante plug-ins muy sencillo de utilizar (ofreciéndonos la posibilidad de usar Spring Boot). Como última nota sobre Eclipse, cabe mencionar que es un programa libre y de código abierto.

Para este proyecto, la instalación de Eclipse será la siguiente:

Descargamos Eclipse de https://www.eclipse.org/downloads la versión más reciente para nuestro sistema operativo (OS), en nuestro caso siendo este macOS. Cuando tengamos el instalador descargado, lo ejecutamos y seleccionaremos Eclipse IDE for Enterprise Java Developers, dado que necesitaremos la herramienta JPA.

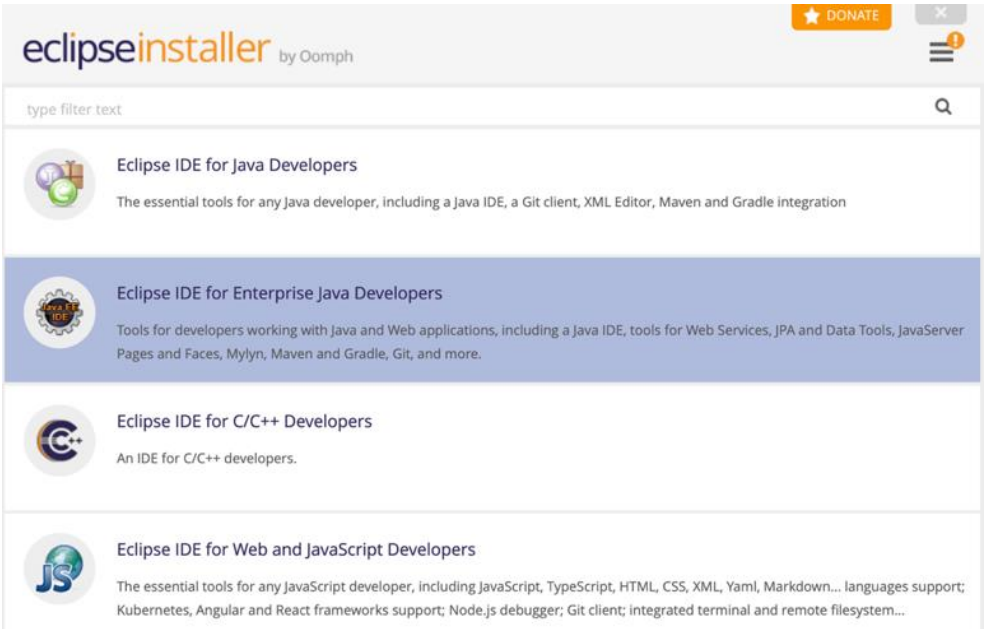

Ilustración 27: Eclipse

Tras seleccionar Enterprise, seleccionamos el directorio en el que queremos realizar la instalación y ya estaremos listos para usar Eclipse.

Visual Studio Code

Usaremos Visual Studio Code para desarrollar el front-end. Visual Studio Code es un editor de código fuente creado por Microsoft y escrito en TypeSript, JavaScript y CSS. Este editor de código es el más popular a nivel mundial y es un programa libre y de código abierto.
Su característica principal es su "agnosticismo" respecto a los lenguajes, lo que significa que podemos usarlo para editar código de cualquier lenguaje, aunque nos ofrece diferentes características en función de cada lenguaje. Esto lo convierte en un editor extremadamente ligero y que para permitirnos desarrollar en cualquier lenguaje nos ofrece la posibilidad de usar extensiones, fácilmente accesibles y pudiendo instalarlas con un solo clic.

Para este proyecto, la instalación de Visual Studio Code será la siguiente:

Nos dirigimos a la página https://code.visualstudio.com/ en la que podemos descargar directamente la versión apta para nuestro OS, que es macOS.

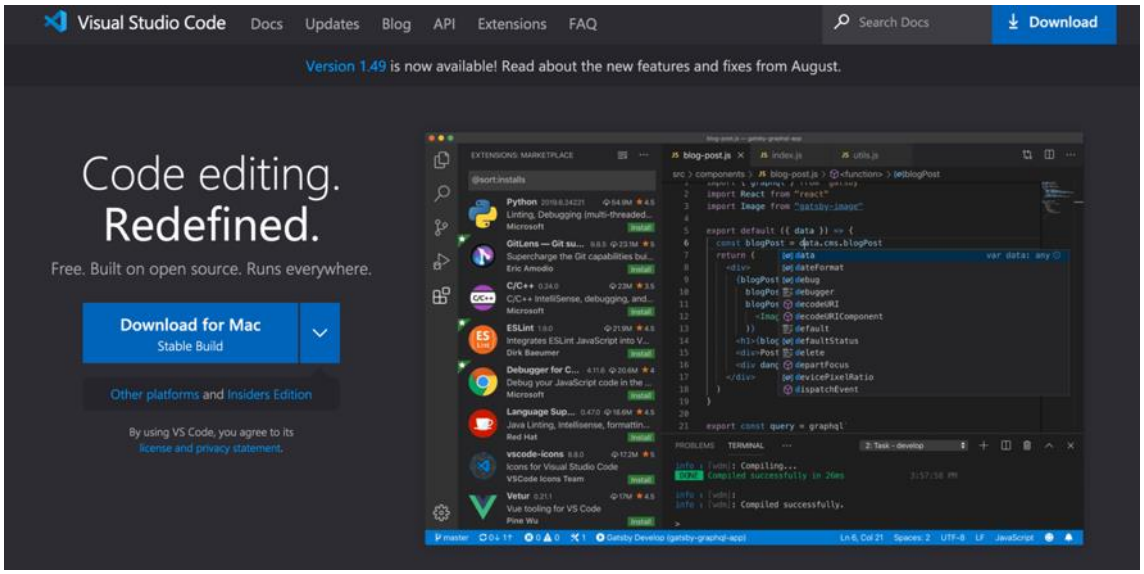

Ilustración 28: Visual Studio Code

Descargamos el zip, lo descomprimimos y movemos la aplicación de Visual Studio Code al directorio que queramos. Tras estos escasos pasos ya estaría lista para poder utilizarla. Esto es gracias a que realmente es un editor súper ligero que podemos extender gracias a las extensiones de su Marketplace.

Spring Boot

Previamente hemos repasado las características de Spring Boot, pero necesitamos poder usarlo para crear nuestro proyecto. Eclipse nos facilita crear aplicaciones de Spring Boot gracias a Spring Tools. Para poder instalar Spring Tools en Eclipse, tenemos que irnos al apartado de ayuda en Eclipse y abrir el Eclipse Marketplace. Una vez allí podemos buscar Spring Tools e instalaremos la versión más reciente.

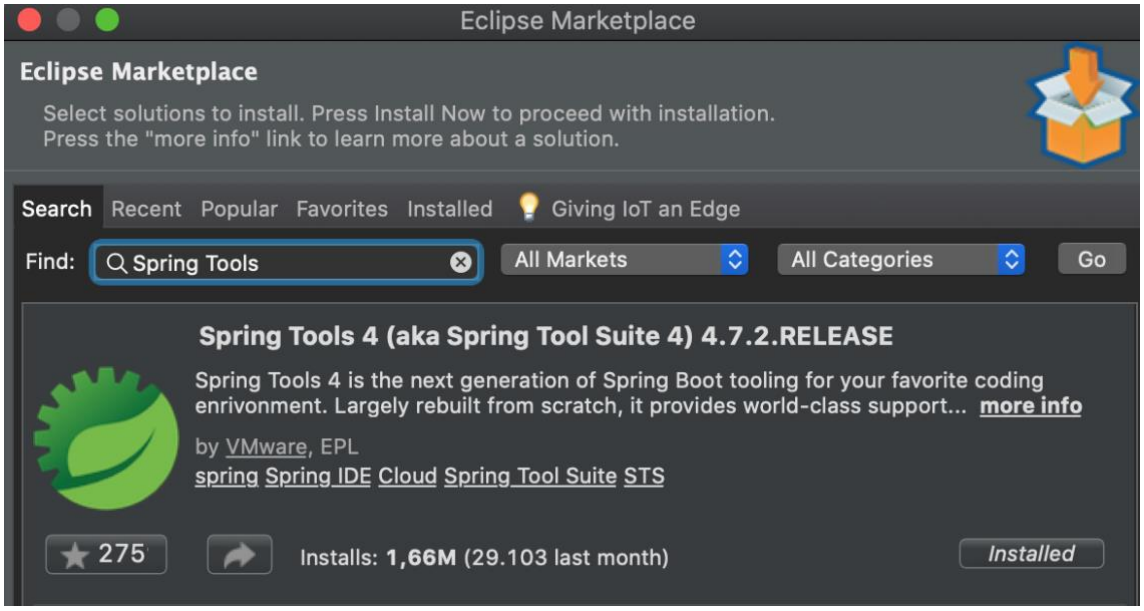

Ilustración 29: Spring Boot

Con darle al botón de instalar, dispondremos del framework Spring Boot para poder crear nuestras aplicaciones en Eclipse.

Angular

Angular (o Angular 2+) es un framework para aplicaciones web escrito en TypeScript. Es de código abierto y fue creado por el equipo Angular de Google. Su característico 2 en el nombre se debe a que es una reescritura del framework en TypeScript de AngularJS. Es el tercer framework front-end más popular según sus estrellas en GitHub, aunque para aprender a usarlo se requiere de un gran esfuerzo y dedicación. Este framework usa los lenguajes de TypeScript, HTML y CSS que nos permiten crear y decorar el front-end de nuestro proyecto. No indagaremos en mucha profundidad en las implementaciones en Angular ya que no es el objetivo de este TFG, pero explicaremos la línea general que sigue el proyecto en el apartado de Implementación. El principal motivo de la elección de este framework para desarrollar el front-end es la familiaridad previa con el mismo, su potencia y extensibilidad gracias a las bibliotecas creadas por la comunidad de usuarios.

Para este proyecto, usaremos angular en Visual Studio Code.

Lo instalamos en el MarketPlace de Visual, seleccionamos la extensión de Angular y la instalamos.

Ilustración 30: Angular

Con estos sencillos pasos estaremos listos para poder crear nuestro proyecto de Angular en Visual Studio Code.

MySQL Workbench

MySQL Workbench es una herramienta visual para el diseño de Bases de Datos. Usaremos la versión de código abierto y gratuita, ya que existe una versión profesional llamada Enterprise pero sus características adicionales no nos interesan. Sus principales funciones son la creación de Bases de Datos MySQL, el mantenimiento y administración de estas, además de facilitarnos visualizar la información contenida en las mismas. También nos permite realizar consultas SQL sobre las instancias que tengamos creadas. Las Bases de Datos MySQL son las segundas más populares mundialmente. MySQL Workbench nos brinda una manera muy sencilla y útil para gestionar las Bases de Datos que tendremos que crear para nuestro proyecto.

Para instalarlo, nos dirigimos a https://www.mysql.com/products/workbench/ donde podremos descargar el archivo .dmg para macOS

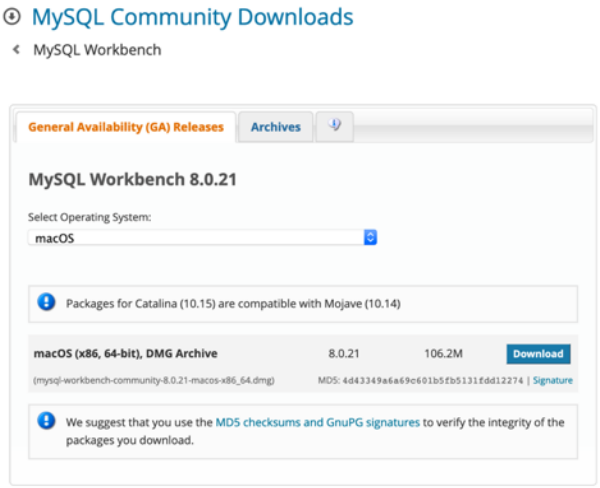

Ilustración 31: MySQL Workbench

Tras la descarga, ejecutamos el archivo .dmg e instalaremos el programa en la carpeta Aplicaciones. Tras este sencillo proceso de instalación, ya podremos crear nuestras instancias propias y administrarlas.

Postman

Postman es una herramienta que nos apoya en el desarrollo de software actuando como un cliente para las APIs REST. Consigue esto permitiéndonos crear peticiones HTTP tanto simples como complejas para poder testear los controladores de nuestro proyecto.

Para descargar Postman, debemos dirigirnos a https://www.postman.com/downloads/ y descargar allí el zip que contiene la aplicación. Con descomprimirlo, podemos mover la aplicación al directorio deseado y empezar a utilizarla.

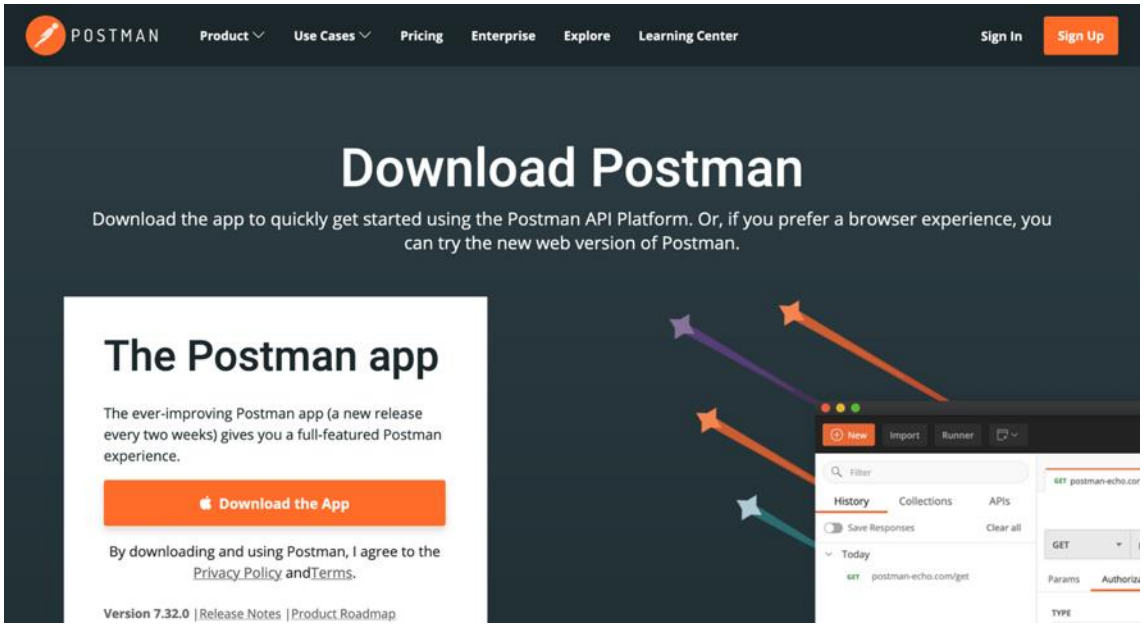

Ilustración 32: Postman

Más adelante veremos ejemplos del uso de Postman, y quedará claro lo sencillo y útil que es realmente este programa para realizar tests.

Arquitectura Software

Para nuestra aplicación decidimos usar un patrón que es prácticamente el estándar a la hora de desarrollar aplicaciones web, que es el patrón Modelo-Vista-Controlador. Lo elegimos porque ofrece una visión mucho más clara del proyecto y sus partes, además de por ser ideal para aplicaciones escalables.

El concepto teórico detrás de este patrón se basa en la separación en capas de la aplicación, la cual se hará función de las responsabilidades que debe tener cada una de ellas:

- El modelo en nuestra aplicación será el Back-End, es decir, los microservicios de Spring Boot. El Back-End se encargará de recuperar los datos de las Bases de Datos y ejecutar la lógica de negocio necesaria para entregar los datos correctamente, según lo que haya solicitado el Controlador.
- La vista en nuestra aplicación será el navegador web y se encargará de mostrar la información de una manera visualmente atractiva para el usuario. Conseguirá la información conectándose con el Controlador para recibirla y la expondrá según le indique.
- El controlador en nuestra aplicación será el Front-End, es decir, la aplicación de Angular. Actuará como intermediario entre el modelo y la vista y procesa las peticiones de la vista para solicitar los datos necesarios al modelo, y posteriormente entregárselos a la vista junto con un estilo de presentación.

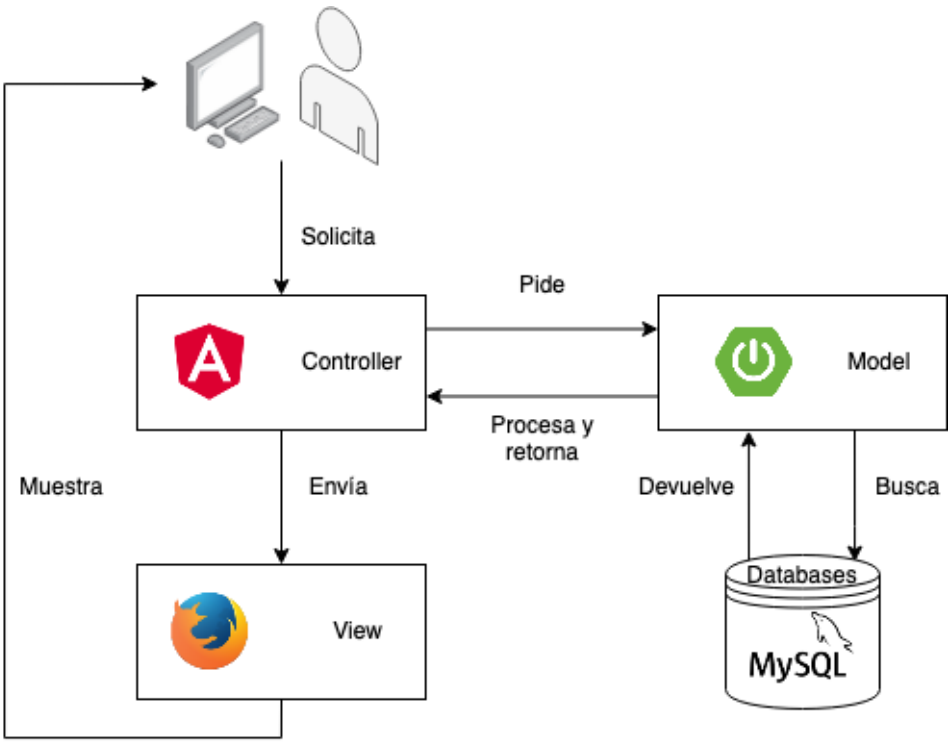

Ilustración 33: Esquema MVC

Tras definir el patrón MVC que seguirá nuestra aplicación, tenemos que abordar cuáles serán los componentes que tendrán el Back-End y el Front-End. La definición de estos es muy sencilla, ya que siguiendo las Historias de Usuario podemos entender fácilmente qué partes debemos desarrollar. Por ejemplo, en la HdU 1 simplemente de un vistazo rápido, deducimos que necesitamos un componente Home para el Front-End y dos componentes para el Back-End, que son Series y Films. Siguiendo este proceso, hemos diseñado el siguiente diagrama para poder visualizar y entender rápidamente qué estructura tendrá nuestra aplicación.

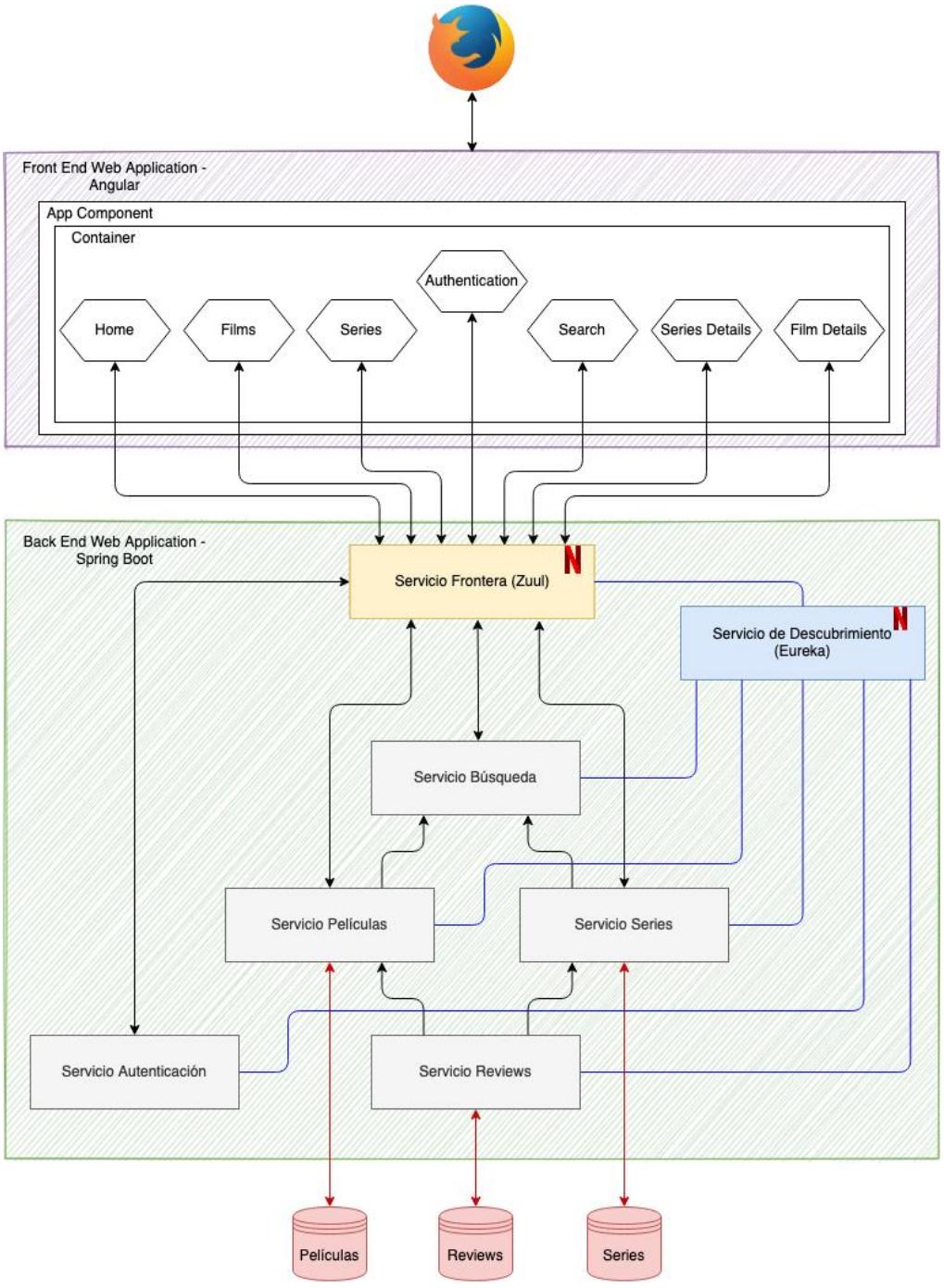

Ilustración 34: Arquitectura Aplicación

Implementación

Una vez definidos los componentes que tendrá nuestra aplicación y siempre teniendo en cuenta las Historias de Usuario, podemos empezar a materializar todas las partes. En este proceso podemos distinguir dos partes claramente diferenciadas, el desarrollo del Back-End y del Front-End. Primero desarrollaremos el Back-End, ya que es la parte central y la gran motivación detrás de este proyecto y al finalizar se seguirá con el Front-End.

Para comenzar con el desarrollo del Back-End revisamos las Historias de Usuario para recoger todos los datos relativos a esta parte de la aplicación. Probablemente es uno de los puntos más críticos del desarrollo, ya que implementar alguna característica mal o no haberla ni definido, nos llevará a perder tiempo y en el mundo laboral además del tiempo, perderemos dinero.

Analizando detenidamente las Historias de Usuario y teniendo en mente la arquitectura que hemos definido del Back-End, estamos preparados para definir los componentes:

Eureka

El componente Eureka es, probablemente, el más importante de todos para nuestra aplicación basada en microservicios. Ya hemos explicado su objetivo, que es el registro de microservicios, pero ahora es momento de definir la implementación de este servicio.

Cuando queremos crear un proyecto de Spring Boot, tenemos a nuestra disposición una herramienta previamente mencionada, que es Spring Initializr. Con esta herramienta podemos crear nuestro proyecto con toda la estructura de carpetas y archivos necesarios para tener un proyecto de Spring Boot siguiendo unos sencillos pasos. En Eclipse, debemos crear un nuevo proyecto y seleccionar la extensión Spring Starter Project de Spring Boot. Una vez lo seleccionemos, tendremos ante nosotros Spring Initializr, donde podremos configurar características tremendamente importantes de nuestro proyecto como la versión de Java, el nombre del proyecto o el grupo al que pertenece. En nuestro proyecto usaremos Java 8 para crear nuestros servicios, y estarán todos en el grupo com.tfg. Además, usarán Maven para definir sus Project Object Model.

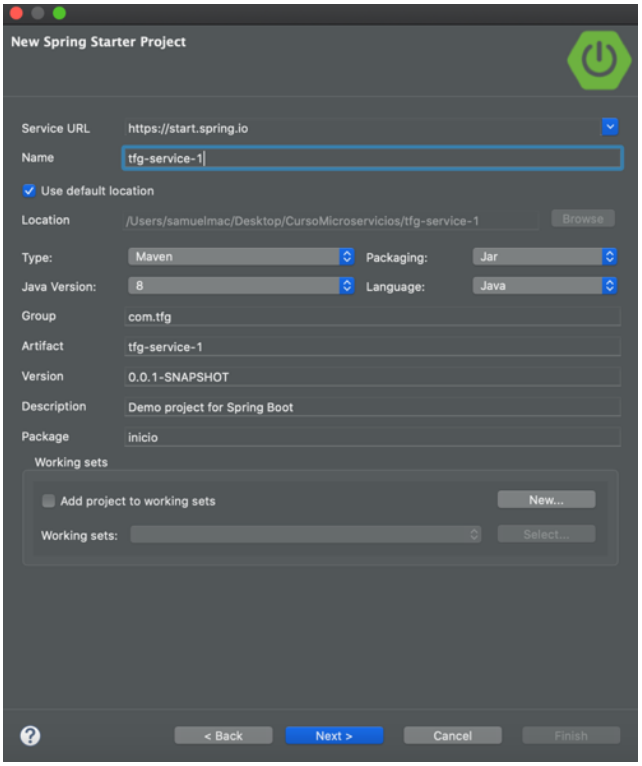

Ilustración 35: Spring Initializr Eureka

Tras definir estas características generales, pasamos a la parte más relevante a la hora de crear nuestro proyecto, que es añadirle las dependencias correctas. Para nuestro proyecto, crearemos el servicio Eureka con las siguientes características:

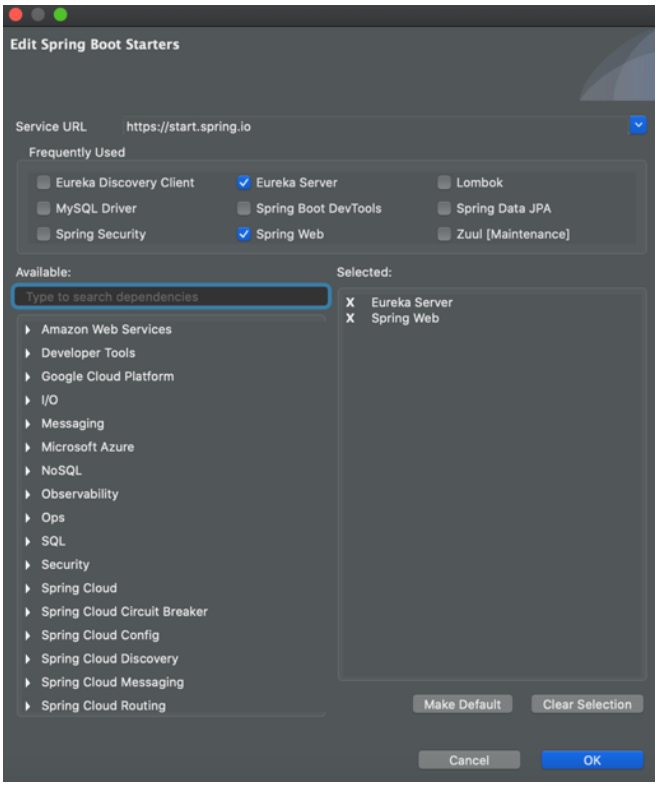

Ilustración 36: Dependencias Eureka

La dependencia de Spring Web nos permite construir aplicaciones RESTful que contengan Spring MVC y que se ejecutan en un servidor Apache Tomcat embebido, que se usa para desplegar el servicio. La dependencia que dará a este servicio su identidad será Eureka Server. Esta dependencia permite a este servicio actuar como un servidor Eureka, si es configurado correctamente. Una vez creado nuestro proyecto, debemos ir a la clase principal y anotarla con @EnableEurekaServer para activar la configuración relativa a Eureka.

```java
package inicio;

import org.springframework.boot.SpringApplication;

@EnableEurekaServer
@SpringBootApplication
public class TfgEurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(TfgEurekaServerApplication.class, args);
    }

}
```

Ilustración 37: Main Eureka

Tras activarla, podemos definir la configuración de nuestro servidor Eureka en el application.yml del proyecto. Lo llamaremos eureka-server, escuchará peticiones en el puerto 8761, donde se registrarán los servicios. Si accedemos a esa URL pero sin /eureka, podremos ver estadísticas del servidor. Además, deshabilitamos su registro con eureka, ya que es el servidor, y le damos una URL por defecto.

```yaml
spring:
  application:
    name: eureka-server
server:
  port: 8761
eureka:
  client:
    register-with-eureka: false
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

Ilustración 38: application.yml Eureka

Tras esta configuración, ya tenemos completamente configurado nuestro servidor Eureka y listo para ser ejecutado. Para ello, debemos ejecutarlo como un proyecto de Spring Boot, y podemos ver los servicios que tiene registrados y demás métricas interesantes. Cuando terminemos el desarrollo mostraremos una captura con todos los servicios levantados y cómo Eureka los tendrá registrados, para que quede realmente claro qué es lo que nos mostrará Eureka en esa dirección.

Zuul

El componente Zuul será el servicio frontera, que a estas alturas debería resultarnos muy familiar. En cuanto a los detalles de su implementación, al igual que Eureka, es un servicio bastante sencillo de configurar gracias a cómo está diseñado. Repetimos el proceso de creación con Spring Initializr, en este caso lo llamaremos tfg-zuul-service. Para que pueda actuar como un servicio Zuul, debemos añadir la dependencia Zuul [Maintenance], y como dependencias adicionales añadiremos Spring Web igual que en Eureka y una nueva dependencia llamada Eureka Discovery Client, que permitirá a este servicio ser registrado por Eureka. En este caso, además añadiremos Spring Security con JWT. Este sistema nos permitirá la autenticación de usuarios mediante JWT, que es un token de acceso creado cuando inicia sesión cada usuario.

Tras crear el proyecto, debemos anotar la clase principal con las anotaciones @EnableZuulProxy y @SpringBootApplication. Estas anotaciones nos permitirán usar todas las características que nos ofrece Zuul, que hemos repasado anteriormente. Para configurarlo, debemos hacerlo en el application.yml del proyecto. Esta configuración será de las más extensas, dado que debemos declarar todos los servicios y las rutas que debemos interceptar. Estas rutas serán las que definiremos en cada servicio respectivamente.

```yaml
spring:
  application:
    name: zuul-service
server:
  port: 7000
zuul:
  routes:
      serv-auth:
        path: /auth/**
        serviceId: auth-service
        sensitive-headers: Cookie,Set-Cookie
        strip-prefix: false
      serv-films:
        path: /films-service/**
        serviceId: films-service
      serv-series:
        path: /series-service/**
        serviceId: series-service
      serv-search:
        path: /search-service/**
        serviceId: search-service
      serv-review:
        path: /review-service/**
        serviceId: review-service
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

Ilustración 39: application.yml Zuul

Además de esta configuración, debemos establecer el sistema de seguridad. Este funcionará interceptando las peticiones a las rutas que define Zuul, y las únicas que permitirá sin identificar será las que se hagan al servicio de autorización, que veremos más adelante. Para configurar esto, debemos crear métodos que nos permitan: aceptar orígenes cruzados, ya que por defecto nuestra aplicación no permitirá esta característica, denegar las peticiones a rutas securizadas (peticiones que no contengan el token de identificación) y permitir el paso de peticiones identificadas por el token. Este token se obtendrá realizando una petición post al servicio de autenticación con unas credenciales correctas, y en la respuesta obtendremos el token. Con ese token, lo añadimos a las cabeceras de las peticiones (que lo haremos en Angular) y podremos acceder a todas las rutas securizadas.

Tras implementar estas características, podemos ejecutar el servidor Zuul y ver cómo Eureka lo registra. Además, podremos comprobar que no podemos acceder a las rutas de los servicios a través del puerto 7000 a menos que iniciemos sesión, lo cual veremos funcionando más adelante cuando desarrollemos el servicio de autenticación.

Auth

El componente de autenticación será el encargado de gestionar el inicio de sesión. Al igual que todos los componentes del Back-End, lo crearemos con Spring Initializr con las dependencias de Eureka Discovery Client, Spring Web y Spring Security. Tras crear el proyecto, debemos implementar el código y las clases que nos permitan validar los inicios de sesión. Usaremos la clase UsernamePasswordAuthenticationFilter de Spring Security, extendiéndola para poder autenticar a los usuarios y en caso de que sea un usuario correcto, nos devolverá el token en la petición Post que hayamos realizado. Los usuarios los definimos por código, por ser más breves en el desarrollo. Su configuración mediante application.properties es muy genérica, únicamente definimos el nombre del servicio, su puerto y la dirección en la que se podrá registrar en Eureka.

```
spring.application.name=auth-service
server.port=9101
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

Ilustración 40: application.yml Auth

Films

El componente Films será el encargado de gestionar las películas. Lo crearemos con Spring Initializr, añadiendo las dependencias Spring Web, Eureka Discovery Client, MySQL Driver que nos permitirá conectarnos con la base de datos Films (que tenemos que haber creado y que contenga la información de las películas) y Spring Data JPA que nos permitirá acceder a la base de datos y recuperar los datos, convirtiéndolos para que Java pueda reconocerlos y operar con ellos. Tras crear el proyecto, creamos las capas de controller, service, dao y model junto con algunas clases, que nos permiten estructurar la aplicación con el patrón MVC y dao, para que cada capa se encargue de una sola

función, las cuales describiremos más adelante. El proyecto debería tener este aspecto a estas alturas.

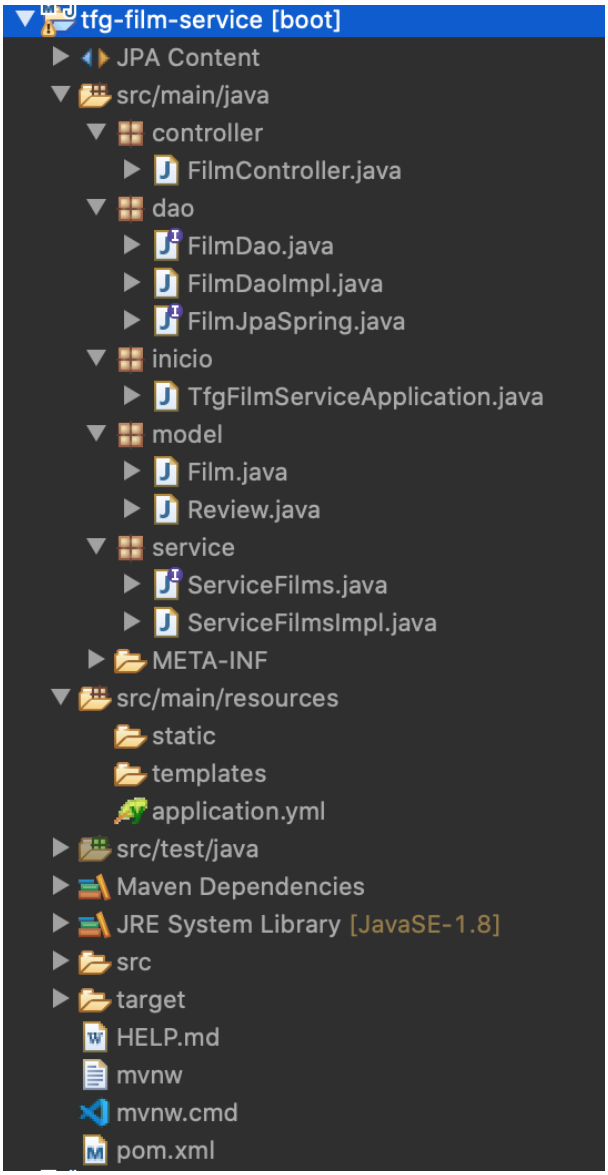

Ilustración 41: Estructura proyecto Films

Para empezar con el desarrollo, debemos definir qué es una película, lo que significa que debemos definir sus campos en el modelo. El modelo se encarga de declarar estos campos, y gracias a la información recogida en las Historias de Usuario y serán los siguientes: un campo idfilm de tipo integer como identificador para la primary key de cada película, un campo duration de tipo integer que defina la duración, un campo genre de tipo cadena que contendrá los géneros de los que sea la película, un campo name de tipo cadena con el nombre de la película, un campo year de tipo integer con el año de estreno, un campo director de tipo cadena con el director de la película, un campo rating de tipo integer con la nota de la crítica, un campo summary de tipo cadena que contenga una breve sinopsis de la película, un campo urlimg de tipo cadena que contenga la referencia a la portada, un campo urltrailer de tipo cadena que contenga la URL del

trailer y un campo reviews de tipo cadena que contenga todas las reviews. Como punto más importante en la capa de model, es que debemos anotar la clase Film.java con @Entity, lo que nos permite que las películas puedan ser manejadas por JPA y podamos tanto recuperarlas de la base de datos como persistirlas. También definiremos el modelo de Review ya que las películas necesitan tenerlas como un campo más, pero lo explicaremos en el servicio Review.

Una vez definidos los campos, nuestro siguiente paso más relevante en el desarrollo es la creación de los endpoints en la capa controller y los métodos de la capa service. En la capa controller definiremos 4 endpoints que nos permitirán acceder a todos los datos que necesitamos para que nuestro servicio de películas funcione según lo definido en las Historias de Usuario: uno para recuperar todas las películas, otro para recuperar una película por su nombre, otro para recuperar una película por su ID y un último para recuperar las películas en función de su género. Es muy importante que anotemos el controller con @RestController, para que pueda usar los métodos REST.

```java
package controller;

import java.util.List;

@CrossOrigin(origins="*")
@RestController
public class FilmController {

    @Autowired
    ServiceFilms service;

    @GetMapping (value ="films", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Film> getFilms () {
        return service.getFilms();
    }

    @GetMapping (value = "films/name/{filmname}", produces = MediaType.APPLICATION_JSON_VALUE)
    public Film getFilmByName(@PathVariable("filmname") String name) {
        return service.getFilmByName(name);
    }

    @GetMapping (value = "films/{filmid}", produces = MediaType.APPLICATION_JSON_VALUE)
    public Film getFilmById(@PathVariable("filmid") Integer id) {
        return service.getFilmById(id);
    }

    @GetMapping (value ="films/genre/{genrename}", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Film> getFilmsByGenre(@PathVariable("genrename") String genre) {
        return service.getFilmsByGenre(genre);
    }

}
```

Ilustración 42: Controller Films

En la capa service es importante que anotemos la clase de implementación con @Service, para que Spring sepa manejar esta Bean. También es importante que nos fijemos en el dao y el resttemplate que definimos. Los dos los anotaremos con autowired para implementar el patrón de DI. En esta clase definiremos la lógica de negocio, es decir, qué transformaciones debemos hacer a los datos recibidos de la consulta a la base de datos. Para recuperar la lista de películas no realizamos ninguna transformación interesante, pero para el resto de los métodos, merece la pena comentar qué es lo que estamos haciendo. Para recuperar las películas por nombre, recuperamos de la capa dao las películas que coincidan con el criterio de búsqueda y, además, le añadiremos las reviews correspondientes a esa película, que las obtendremos mediante una petición con el resttemplate definido y la URL del servicio, porque necesitamos mostrar los comentarios cuando accedemos a una película. Este método se comunica con el servicio

de reviews y le pedirá las reviews respectivas a la película. En el método para recuperar las películas por género, recuperamos todas las películas y las filtramos por género, para poder encontrar coincidencias en las películas que tengan más de un género, cosa que si lo hiciésemos mediante una consulta a la base de datos no sería posible. Por último, el método para recuperar una película por ID es prácticamente igual al de recuperar por nombre.

```java
package service;

import java.util.Collections;

@Service
public class ServiceFilmsImpl implements ServiceFilms{

    @Autowired
    FilmDao dao;

    @Autowired
    RestTemplate template;

    String urlReviews = "http://review-service/review/";

    @Override
    public List<Film> getFilms() {
        List<Film> films = dao.getFilms();
        return films;
    }

    @Override
    public Film getFilmByName(String name) {
        Film film = dao.getFilmByName(name);
        String reviews = template.getForObject(urlReviews + film.getName() , String.class);
        film.setReviews(reviews);
        return film;
    }

    @Override
    public List<Film> getFilmsByGenre(String genre) {
        List<Film> films = dao.getFilms();
        films = films.stream()
                .filter(o->o.getGenre().contains(genre))
                .collect(Collectors.toList());
        Collections.shuffle(films, new Random(System.nanoTime()));
        return films;
    }

    @Override
    public Film getFilmById(Integer id) {
        Film film = dao.getFilmById(id);
        String reviews = template.getForObject(urlReviews + film.getName() , String.class);
        film.setReviews(reviews);
        return film;
    }



}
```

Ilustración 43: Service Films

En la capa dao, debemos definir la manera en la que nos comunicamos con la base de datos. Tenemos dos clases importantes, la implementación del dao, donde haremos las llamadas a los métodos de la clase FilmJpaSpring y la clase FilmJpaSpring. Esta clase se encarga de definir cómo serán los objetos que tenemos que recuperar de la base de datos, que serán de tipo Film y su primary key o identificador, que será de tipo integer. Es importante que anotemos la clase dao con @Repository para que Spring sepa identificar esta Bean y la clase FilmJpaSpring debe extender JpaRepository.

```java
package dao;

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

public interface FilmJpaSpring extends JpaRepository<Film,Integer>{

    Film findByIdfilm(Integer idfilm);
    Film findByName(String name);

}

```

Ilustración 44: DAO Films

```java
package dao;

import java.util.List;

@Repository
public class FilmDaoImpl implements FilmDao {
    @Autowired
    FilmJpaSpring films;

    @Override
    public List<Film> getFilms() {
        return films.findAll();
    }

    @Override
    public Film getFilmByName(String name) {
        return films.findByName(name);
    }

    @Override
    public Film getFilmById(Integer id) {
        return films.findByIdfilm(id);
    }

}
```

Ilustración 45: DAO Implementation Films

Por último, para la configuración del servicio debemos definir cómo se conectará con la base de datos, la estrategia para la identificación de las entidades de la base de datos, el nombre del servicio, su puerto y su conexión con Eureka.

```yaml
spring:
  application:
    name: films-service
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/films?serverTimezone=UTC
    username: root
    password: 62216221
  jpa:
    hibernate:
      naming:
        implicit-strategy: org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
server:
  port: 8001
# Config de el servidor
eureka:
  client:
    serviceUrl:
      # Direccion a la que se conecta el microservicio con eureka
      defaultZone: http://localhost:8761/eureka
```

Ilustración 46: application.yml Films

Series

El componente Series, muy similar al componente Films, será el encargado de gestionar las series. Como todos los servicios, lo crearemos con Spring Initializr, añadiendo las dependencias Spring Web, Eureka Discovery Client, MySQL Driver que nos permitirá conectarnos con la base de datos Series (que tenemos que haber creado y que contenga la información de las series) y Spring Data JPA que nos permitirá acceder a la base de datos y recuperar los datos, convirtiéndolos para que Java pueda reconocerlos y operar con ellos. Tras crear el proyecto, creamos las capas de controller, service, dao y model junto con algunas clases, que nos permiten estructurar la aplicación con el patrón MVC y dao. Omitiremos la estructura por no redundar demasiado en la explicación de los componentes.

Para empezar con el desarrollo, debemos definir qué es una serie, lo que significa que debemos definir sus campos en el modelo. El modelo se encarga de declarar estos campos, y gracias a la información recogida en las Historias de Usuario y serán los siguientes: un campo idseries de tipo integer como identificador para la primary key de cada serie, un campo duration de tipo integer que defina la duración de cada capítulo, un campo genre de tipo cadena que contendrá los géneros de los que sea la serie, un campo name de tipo cadena con el nombre de la serie, un campo year de tipo integer con el año de estreno, un campo episodes de tipo integer con los episodios de cada temporada, un campo seasons de tipo integer con las temporadas de la serie, un campo rating de tipo integer con la nota de la crítica, un campo summary de tipo cadena que contenga una breve sinopsis de la serie, un campo urlimg de tipo cadena que contenga la referencia a la portada, un campo urltrailer de tipo cadena que contenga la URL del tráiler y un campo reviews de tipo cadena que contenga todas las reviews. Al igual que las películas, accedemos a ellas gracias al Jpa de Spring.

Una vez definidos los campos, nuestro siguiente paso más relevante en el desarrollo es la creación de los endpoints en la capa controller y los métodos de la capa service. En la capa controller definiremos 4 endpoints que nos permitirán acceder a todos los datos que necesitamos para que nuestro servicio de series funcione según lo definido en las Historias de Usuario: uno para recuperar todas las series, otro para recuperar una serie por su nombre, otro para recuperar una serie por su ID y un último para recuperar las series en función de su género. Usaremos las mismas anotaciones que en la capa controller de películas.

En la capa service, definimos las mismas anotaciones y el template que en la capa service de películas. Para recuperar la lista de series no realizamos ninguna transformación interesante. Para recuperar las series por nombre, recuperamos de la capa dao las series que coincidan con el criterio de búsqueda y, además, le añadiremos las reviews correspondientes a esa serie, que las obtendremos mediante una petición con el resttemplate definido y la URL del servicio, porque necesitamos mostrar los comentarios cuando accedemos a una serie. Este método se comunica con el servicio de reviews y le pedirá las reviews respectivas a la serie. En el método para recuperar las series por género, recuperamos todas las series y las filtramos por género, para poder encontrar coincidencias en las series que tengan más de un género, cosa que si lo hiciésemos

mediante una consulta a la base de datos no sería posible. Por último, el método para recuperar una serie por ID es prácticamente igual al de recuperar por nombre.

La capa dao será prácticamente igual que la del dao de películas, así que omitiremos su explicación ya que no nos aportará nada nuevo.

Por último, para la configuración del servicio debemos definir cómo se conectará con la base de datos, la estrategia para la identificación de las entidades de la base de datos, el nombre del servicio, su puerto y su conexión con Eureka.

```yaml
spring:
  application:
    name: series-service
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/series?serverTimezone=UTC
    username: root
    password: 62216221
  jpa:
    hibernate:
      naming:
        implicit-strategy: org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
server:
  port: 8000
# Config de el servidor
eureka:
  client:
    serviceUrl:
      # Direccion a la que se conecta el microservicio con eureka
      defaultZone: http://localhost:8761/eureka
```

Ilustración 47: application.yml Series

Search

El componente Search se encargará de gestionar las búsquedas que realicemos en la aplicación. Lo crearemos con Spring Initializr con las dependencias de Spring Web y Eureka Discovery Client únicamente.

Este servicio es interesante por el hecho de que depende de otros para funcionar. Tendremos que hacer peticiones GET a los servicios Films y Series para poder cribar las búsquedas según los criterios de búsqueda. El servicio estará estructurado con las capas de model, que contendrá una entidad que recoge las características comunes de series y películas, diferenciándolas en función de sus identificadores originales, la capa service en la que realizaremos las peticiones a los servicios films y series y haremos las transformaciones pertinentes y la capa controller en la que expondremos los métodos de la búsqueda.

```java
package service;

import java.util.Arrays;

@Service
public class SearchServiceImpl implements SearchService {

    @Autowired
    RestTemplate template;
    String urlFilm = "http://films-service";
    String urlSeries = "http://series-service";

    @Override
    public List<ObjectSearch> getResults(){
        ObjectSearch[] films = template.getForObject(urlFilm + "/films", ObjectSearch[].class);
        ObjectSearch[] series = template.getForObject(urlSeries + "/series", ObjectSearch[].class);
        ObjectSearch[] resultado = (ObjectSearch[]) ArrayUtils.addAll(films, series);
        return Arrays.asList(resultado);
    }

    @Override
    public List<ObjectSearch> getFiltered(String search){
        return getResults()
                .stream()
                .filter(o->o.getGenre().contains(search)||o.getName().contains(search))
                .collect(Collectors.toList());
    }

}
```

Ilustración 48: Service Search

En esta captura del código de la capa service, podemos observar que el método getResults() hará las peticiones a Films y Series. El método getFiltered() llamará al método getResults() y filtrará las películas y series en función de si coinciden sus géneros y sus títulos con la cadena de búsqueda.

Es importante que nos fijemos en el RestTemplate que usamos en esta clase, ya que gracias a él podremos realizar peticiones a otros servicios. Este lo creamos en la clase main, y lo anotaremos como un Bean para que Spring Boot pueda inyectarlo en el servicio.

```java
package inicio;

import org.springframework.boot.SpringApplication;

@ComponentScan(basePackages= {"controller","service"})
@SpringBootApplication
public class TfgGalleryServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(TfgGalleryServiceApplication.class, args);
    }

    @Bean(value = "restTemplate")
    @LoadBalanced
    public RestTemplate template() {
        return new RestTemplate();
    }
}

```

Ilustración 49: Main Search

Como último aspecto relevante tenemos la capa controller, en la que definimos los endpoints a través de los que expondremos los recursos de este servicio, que únicamente será la función de búsqueda.

```java
package controller;

import java.util.List;

@RestController
public class SearchController {

    @Autowired
    SearchService service;

    @GetMapping(value="{datatosearch}", produces=MediaType.APPLICATION_JSON_VALUE)
    public List<ObjectSearch> getGenre(@PathVariable("datatosearch") String search) {
        return this.service.getFiltered(search);
    }


}
```

Ilustración 50: Controller Search

Review

El componente Review se encargará de gestionar las reviews de tanto películas como series. Estas reviews poseen un integer idreview que funcionará como su identificador, un campo entero rating que contendrá la nota, un campo cadena name que contendrá nombre del título sobre el que se está haciendo la review, un campo cadena user que contendrá el nombre de usuario, un campo texto que contendrá el comentario de la review y un campo integer que indicará si la review almacenada es sobre una película o sobre una serie.

Este servicio lo hemos creado con Spring Initializr y tiene la misma arquitectura que el resto de los servicios que disponen de acceso a base de datos. En la capa model definiremos el modelo descrito previamente para las reviews. En la capa dao, accedemos a los datos de la base de datos reviews mediante JPA, ofreciendo un método que recupere todas las reviews y otro que recupere según el título de la película o serie. En la capa service, simplemente llamaremos a la capa dao y ofreceremos estos datosa la capa controller. En la capa controller expondremos los recursos mediante dos endpoints. Como configuración del servicio, dispondremos de los siguientes parámetros:

```yaml
spring:
  application:
    name: review-service
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/reviews?serverTimezone=UTC
    username: root
    password: 62216221
  jpa:
    hibernate:
      naming:
        implicit-strategy: org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
server:
  port: 8002
# Config de el servidor
eureka:
  client:
    serviceUrl:
      # Direccion a la que se conecta el microservicio con eureka
      defaultZone: http://localhost:8761/eureka
```

Ilustración 51: application.yml Review

Despliegue de prueba Back-End

Habiendo implementado y configurado todos los servicios, es hora de probar que está funcionando todo según lo esperado. Nuestro primer paso será desplegar Eureka, después, el resto de microservicios. Una vez hecho esto, podemos ver que el servidor Eureka tiene registrados a todos los microservicios.

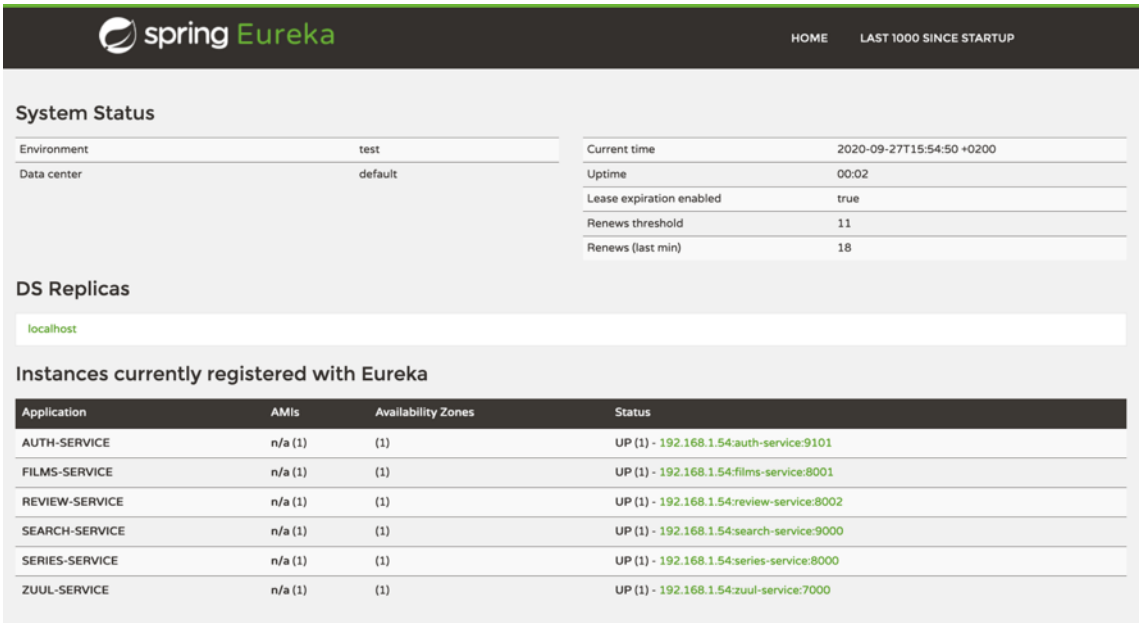

Ilustración 52: Eureka status

Nuestra herramienta para las pruebas será Postman, así que para empezar debemos realizar el inicio de sesión de la siguiente manera:

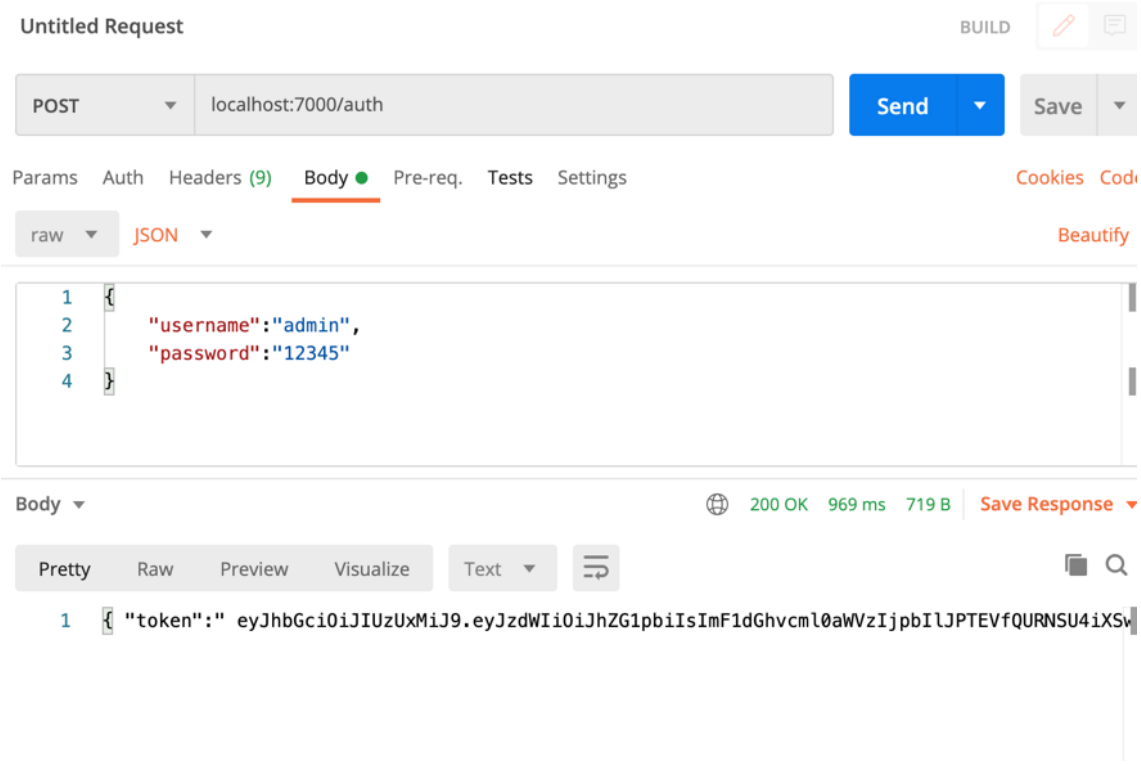

Ilustración 53: Inicio de sesión con Postman

Como podemos observar, nuestro inicio de sesión es correcto (200), así que el servicio auth nos devolverá el token, que debemos añadir a las cabeceras de nuestras peticiones. Aquí podemos ver cómo se haría:

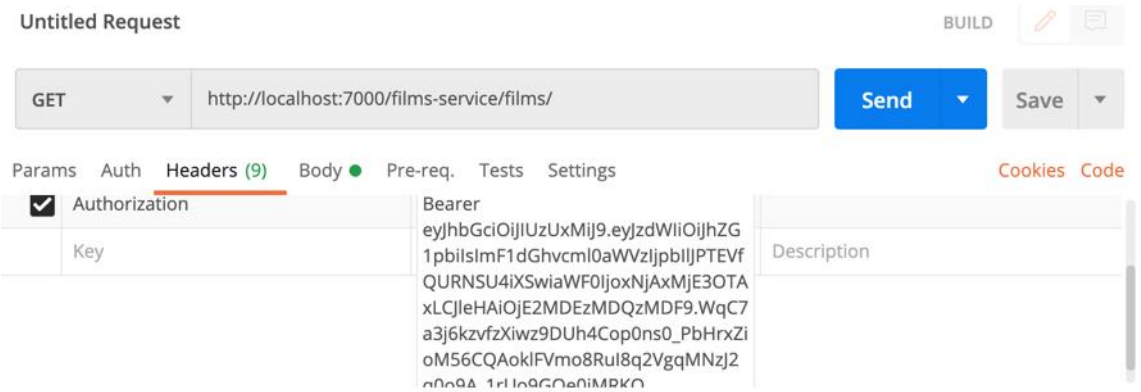

Ilustración 54: Añadiendo como Header el token JWT a una petición

Tras ponerlo, hacemos nuestra petición GET y al estar autorizados, Zuul nos permite acceder al servicio de películas y nos devolverá la lista de todas las películas:

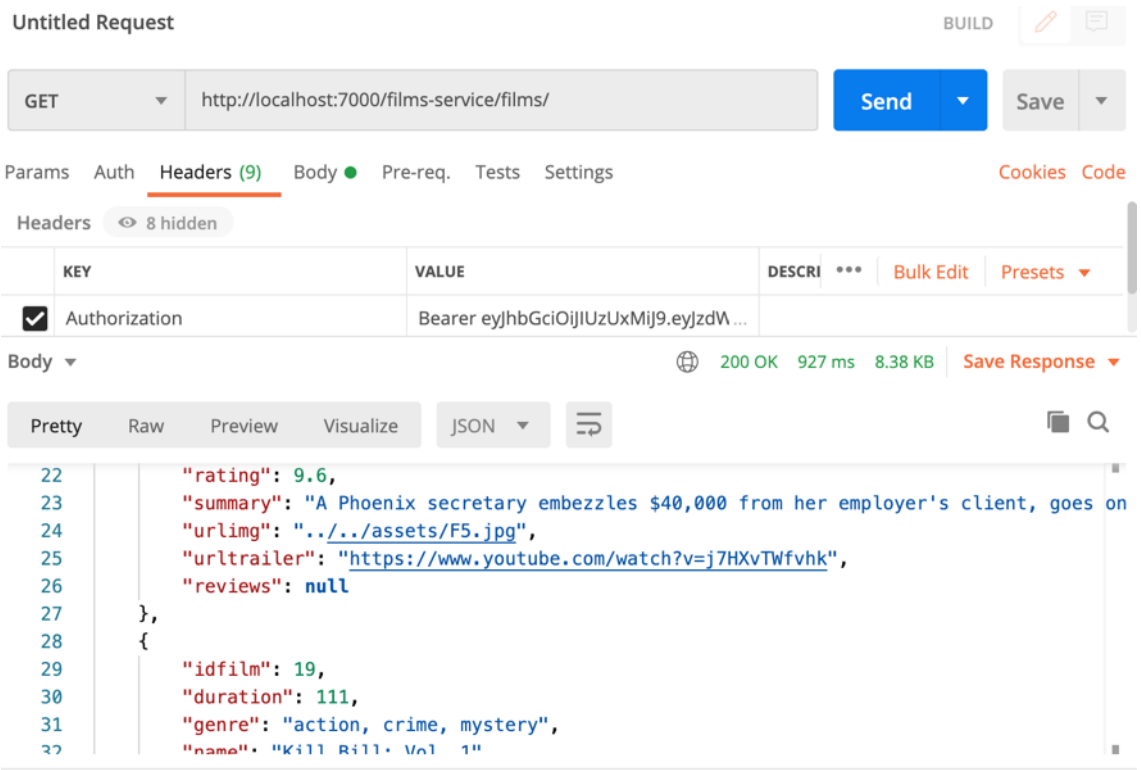

Ilustración 55: Usando el token JWT con Postman

Angular

Una vez revisados todos los componentes del Back-End y probado que funcionan, podemos implementar el Front-End. Para comenzar con el desarrollo revisamos de nuevo las Historias de Usuario para recoger todos los datos relativos a esta parte de la aplicación. Mirando el diagrama de la aplicación, vemos los diferentes componentes que va a tener nuestra aplicación de Angular, así que podemos ponernos a ello.

La arquitectura de nuestra aplicación Angular seguirá el patrón de contenedor. Esto significa que definimos un componente contenedor sobre el que mostraremos el resto de los componentes, dependiendo de la ruta que solicitemos. Estos serán los siguientes: Authentication, Home, Film, Film-details, Series, Series-details y Search. Estos componentes harán uso de servicios que definamos, para conectar con el Back-End. Cada uno de ellos está dedicado para cada microservicio (menos para Zuul y eureka), aunque hay uno extra, que se encargará de interceptar todas las peticiones http (menos las que hagamos al servicio de autenticación) para añadirles el token de inicio de sesión, permitiéndonos que el Front-End pueda realizarlas. En el manual de Usuario podremos ver el resultado de este desarrollo.

Manual de Usuario

En este apartado vamos a detallar cuáles son las funcionalidades de nuestra aplicación y cómo aprovecharlas mediante su interfaz visual. Su diseño se ha inspirado en las diferentes plataformas de streaming online, refinando algunas características según criterios propios y adecuándolas al propósito del proyecto. Cabe mencionar que, si revisamos las HdU, veremos que todas las metas marcadas están totalmente finalizadas y aceptadas siguiendo los criterios de aceptación.

Como método de despliegue, debemos ejecutar primero el servicio Eureka, y cuando esté levantado, podremos levantar todos los servicios restantes, sin importar el orden. En nuestro caso, los hemos ejecutado desde el IDE de Eclipse, como aplicaciones de Spring Boot. Además, debemos ejecutar la aplicación frontend de Angular, para ello, con estar en el directorio de esta e introducir el comando ng serve –open, la aplicación automáticamente abrirá una pestaña en el navegador con la aplicación. Es indispensable contar con las herramientas descritas previamente para realizar el despliegue.

Inicio de sesión

Será la primera página que nos encontremos al iniciar la aplicación, en la que se nos piden unas credenciales de inicio de sesión. Los usuarios han sido codificados dentro del servicio de tfg-auth-service, así que tendremos que usar alguno de los que están allí creados, por ejemplo, Username: Samu, Password: 12345. Si nuestro inicio de sesión es incorrecto, no avanzará a la aplicación hasta que pongamos las credenciales correctas.

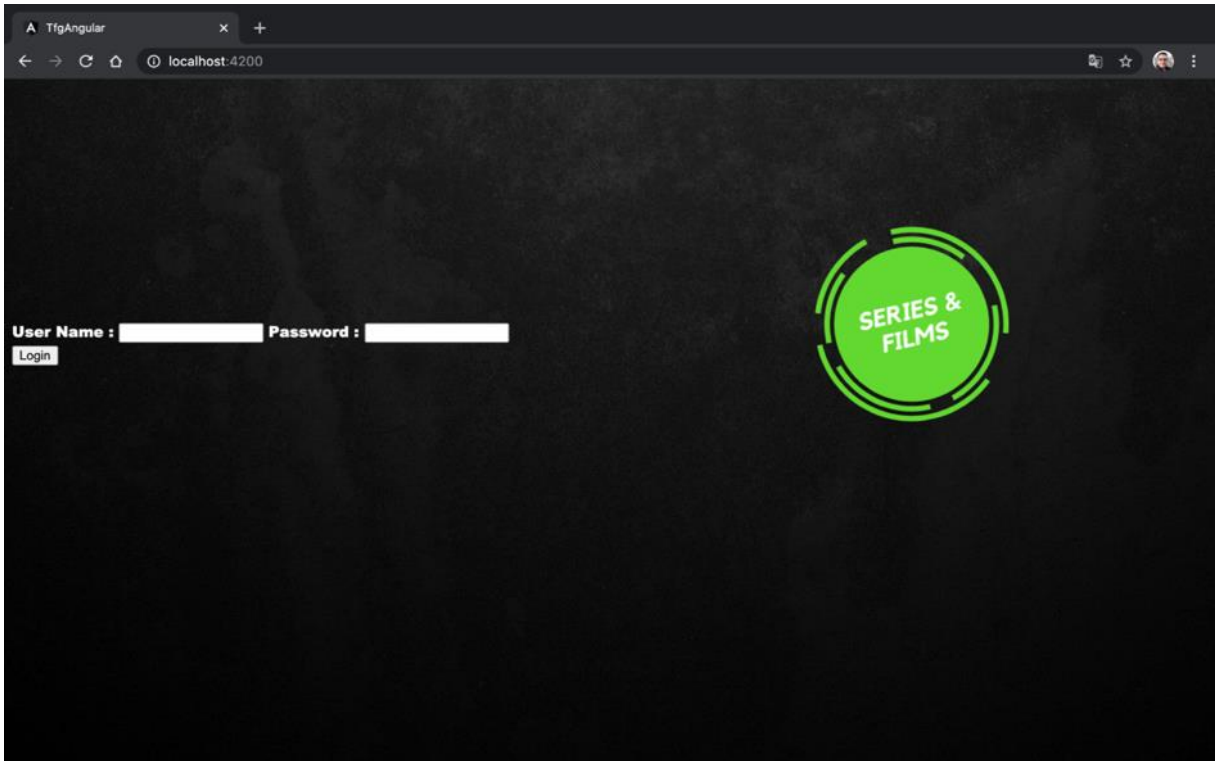

Ilustración 56: Login Angular

Una vez hayamos iniciado sesión correctamente, se mostrará la página principal de la aplicación.

Home

Esta página es el elemento central de la aplicación. Está pensada para que los usuarios puedan ver en ella cualquier tipo de contenido. Por ello, se muestran todas las películas y las series. Los títulos se muestran en listas de 4 en 4, en las que veremos los títulos según su imagen. Estas listas si pulsamos el botón tanto izquierda como derecha, se moverán y mostrarán otros títulos. Podemos pulsar cuantas veces queramos un botón en una dirección y no llegaremos al final de la lista, ya que los títulos se muestran en un bucle infinito. Cuando pulsemos cualquier título, sea película o serie, nos llevará a los detalles de este. Más adelante explicaremos en detalle en qué consiste el apartado de detalles. Además, para facilitar la decisión sobre qué titulo ver la información, tenemos listas en las que se muestran los top 10 de películas y series respectivamente, ordenados por la nota de la crítica. También podemos ver en esta página la barra de navegación de la aplicación, que estará presente en todo momento en la parte superior del navegador, para poder elegir a qué parte de la aplicación podemos ir. Por último, en la misma barra de navegación podemos ver nuestro nombre de usuario, que se mantendrá mientras tengamos la sesión activa.

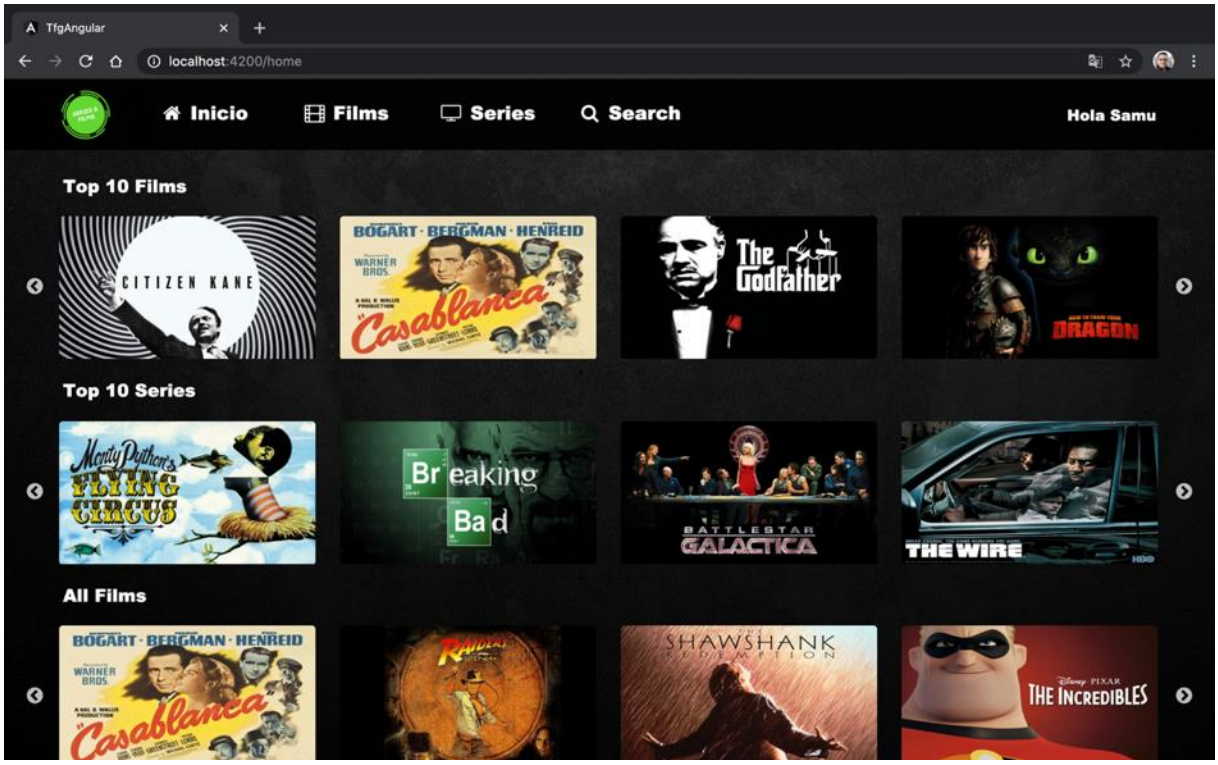

Ilustración 57: Home Angular

Films

En este componente podemos visualizar toda la información relativa a las películas. Vemos que en primer lugar tenemos una lista en la que aparecen todas las películas, por si queremos visualizar qué nos ofrece la aplicación. Debajo de esta lista, hay varias listas que mostrarán las películas clasificadas por género. Las listas siguen el mismo principio que en la página inicial, mostrando los títulos en un bucle infinito. También, cuando pulsemos un título, la aplicación nos redigirá a los detalles de este.

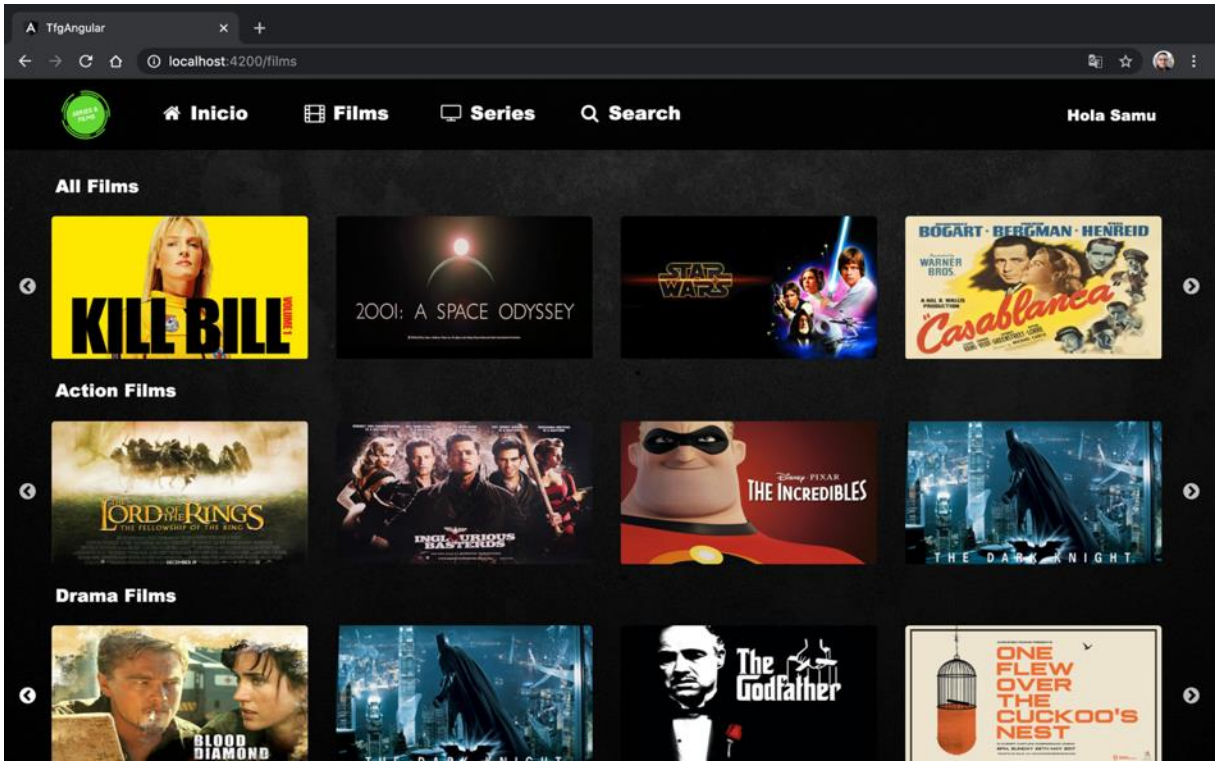

Ilustración 58: Films Angular

Series

En la misma tónica que el componente películas, tendremos una lista que muestre todas las series y debajo una serie de listas en las que podremos ver las series organizadas por su género. Si pulsamos un título, iremos los detalles de esa serie.

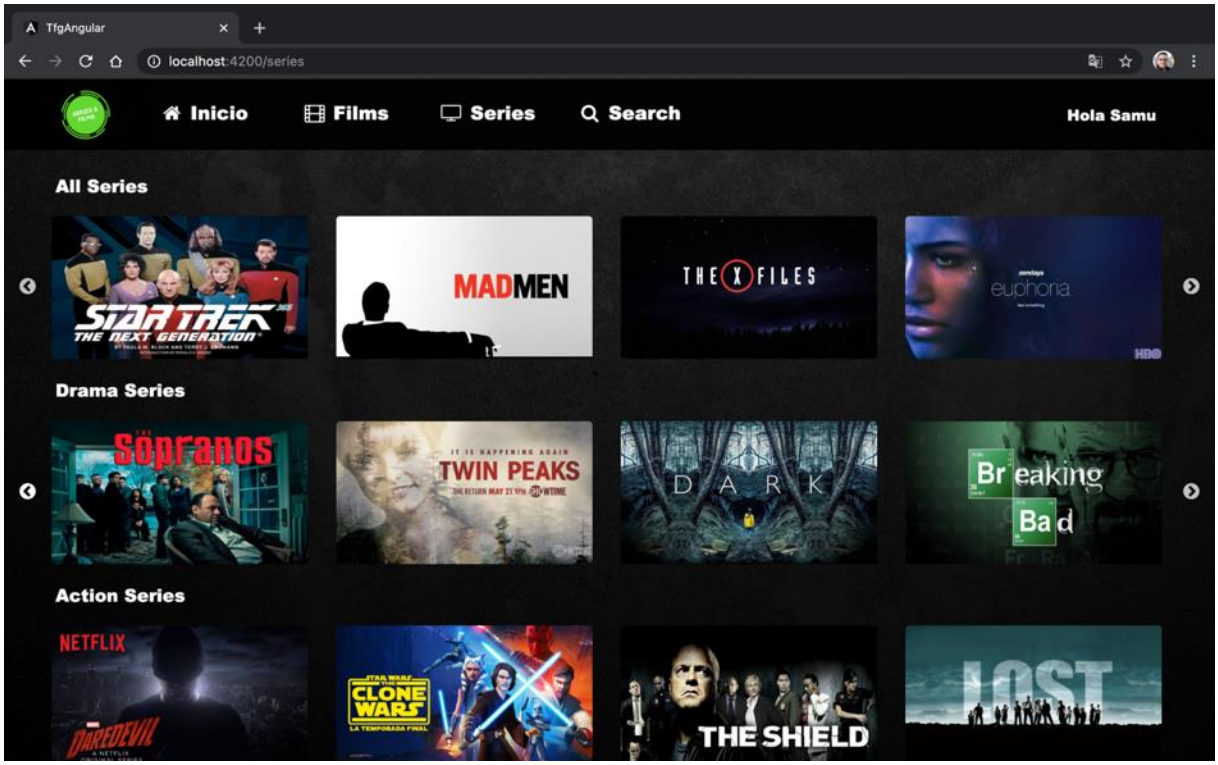

Ilustración 59: Series Angular

Search

La búsqueda en primer momento aparecerá vacía, en la que sólo podremos ver una barra en la que tendremos que introducir nuestra búsqueda. Según introduzcamos caracteres, se realizará la búsqueda, actualizándose con cada pulsación de tecla. Estas se mostrarán en una matriz de 4 columnas y las filas variarán en función del número de resultados de la búsqueda. Siguiendo el propósito de la búsqueda, cuando pulsemos un título, sea película o serie, nos llevará a los detalles del mismo.

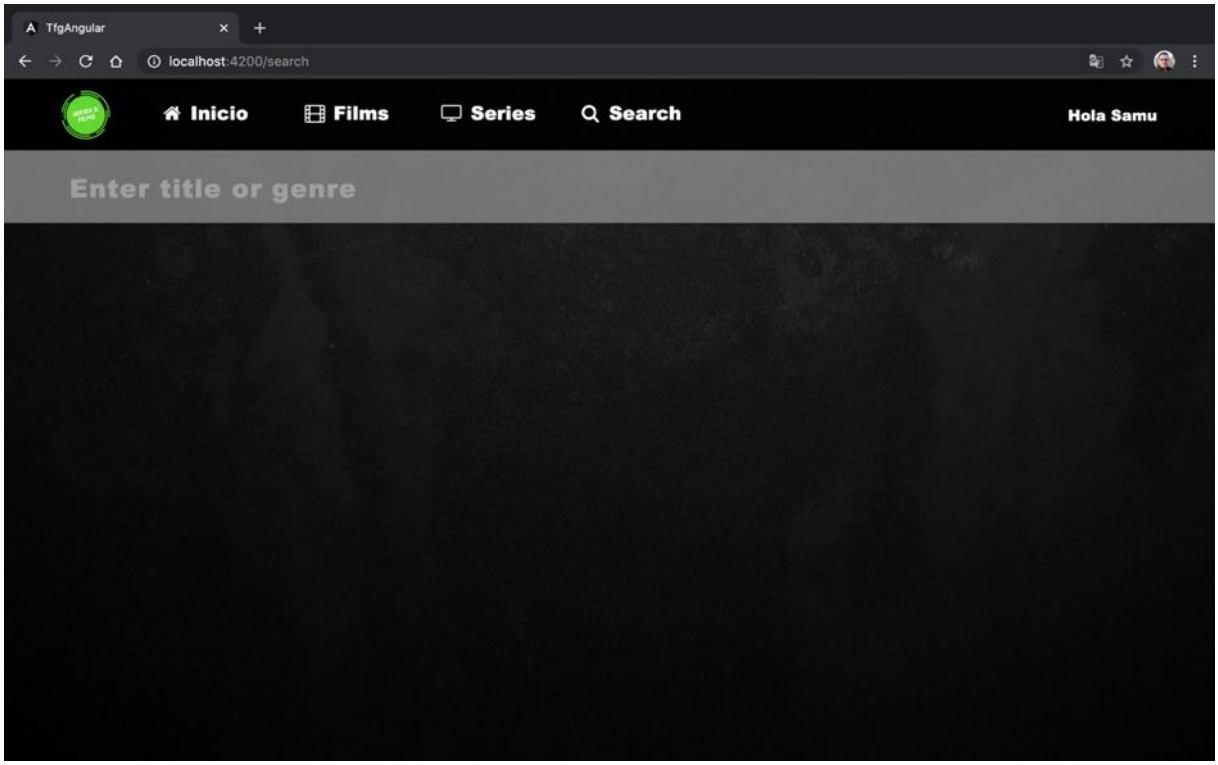

Ilustración 60: Search Angular

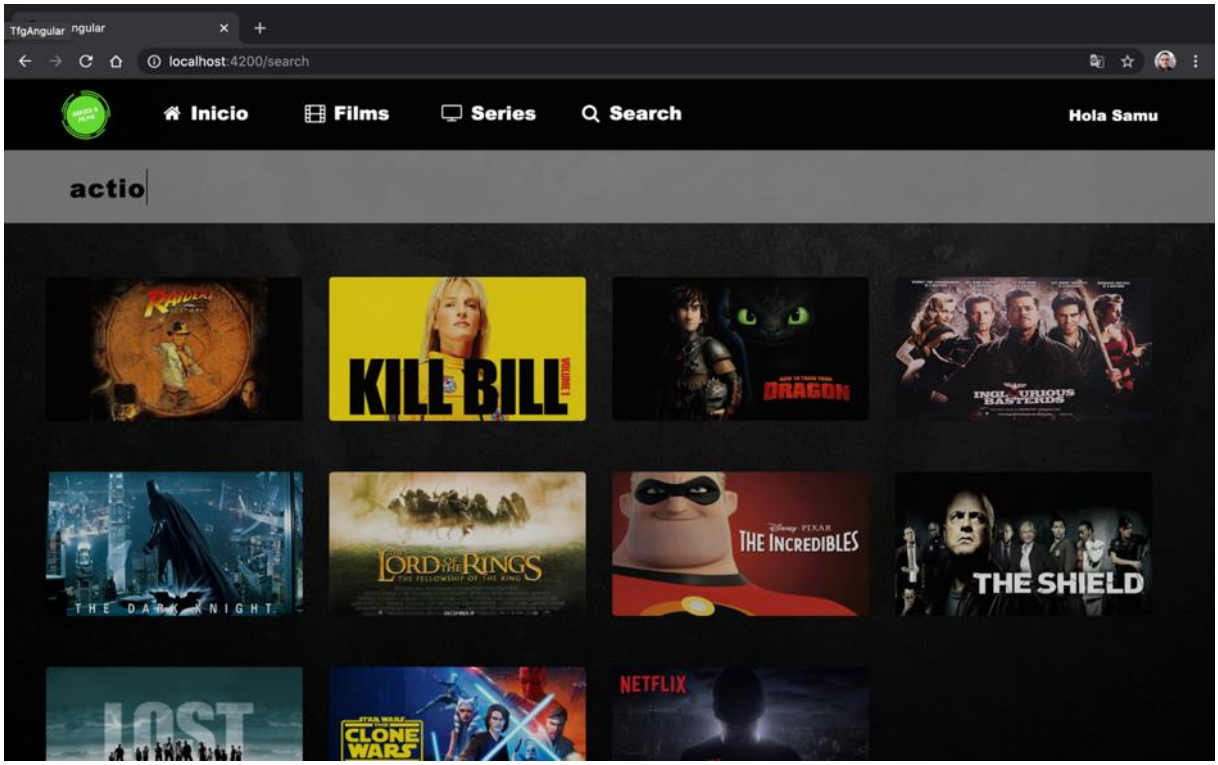

Ilustración 61: Search género Angular

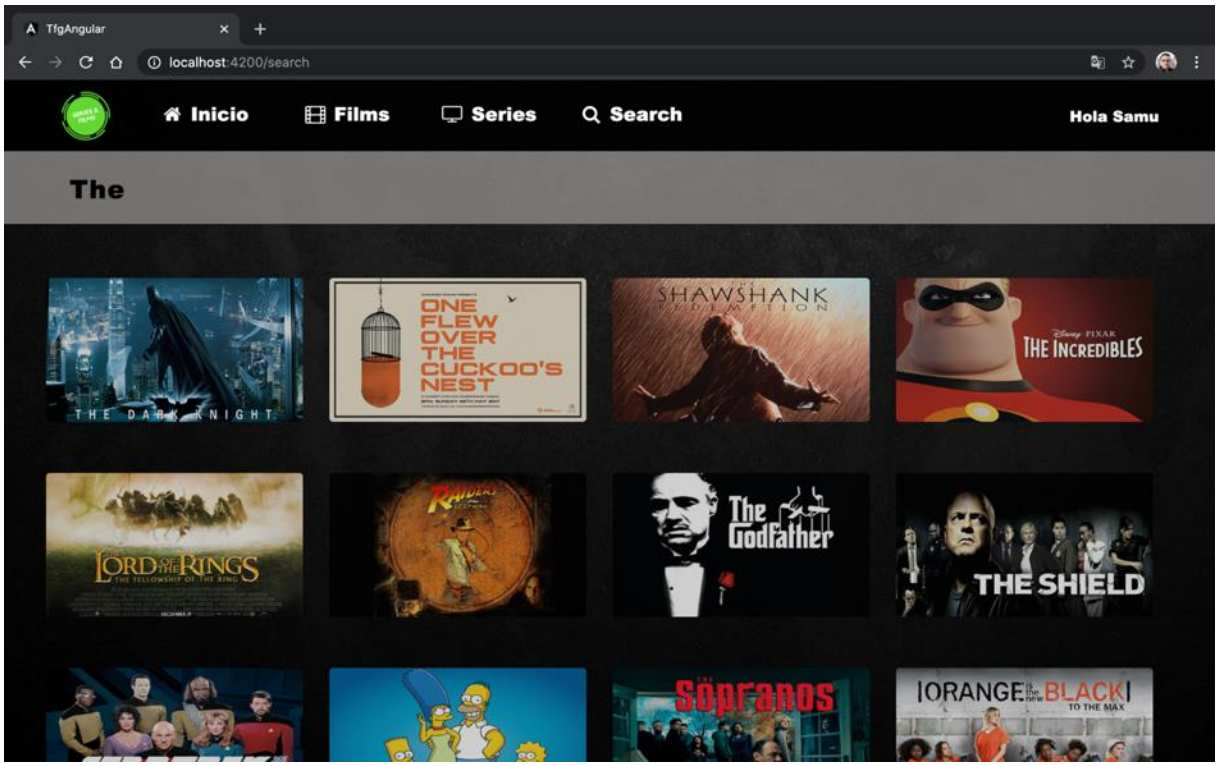

Ilustración 62: Search nombre Angular

Film-details

Cuando hayamos pulsado una película, se nos redirigirá a este componente. Podemos visualizar la portada, una sinopsis de la película, sus respectivas características y un botón que nos llevará al tráiler de esta en YouTube. También veremos debajo de esta su propia sección de comentarios, que detallaremos en su propio apartado.

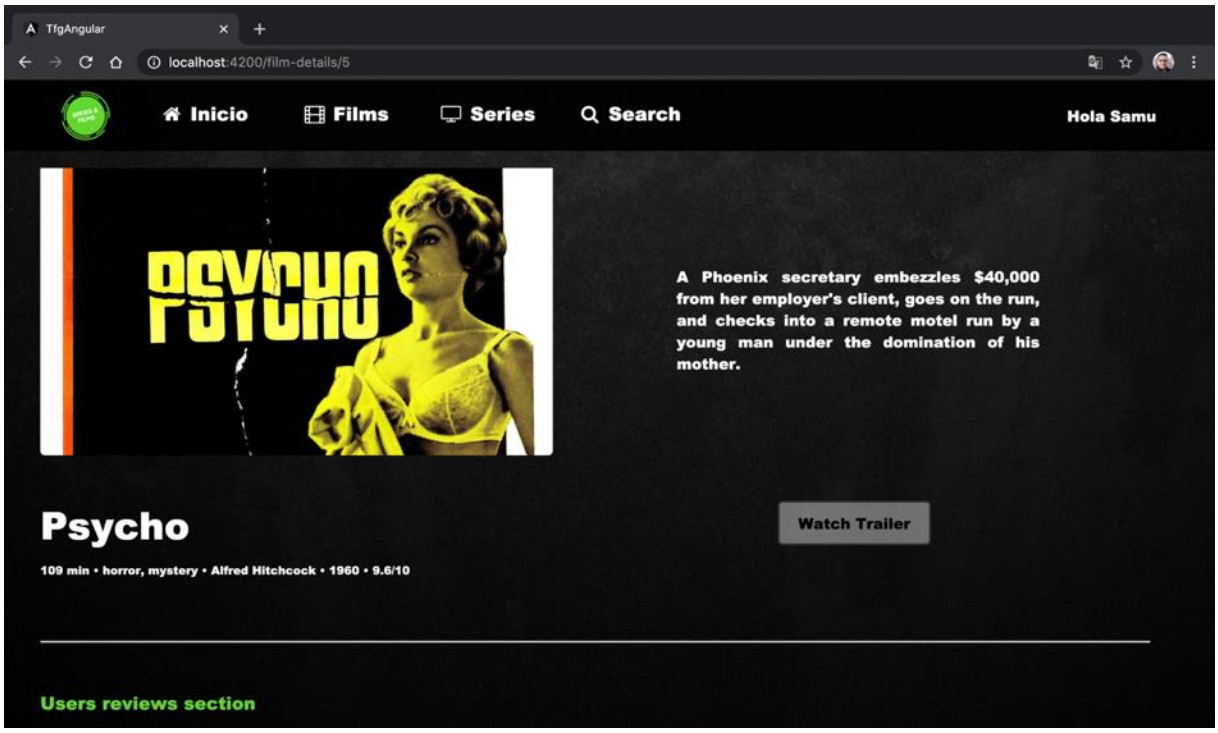

Ilustración 63: Film-details Angular

Series-details

Prácticamente en la misma línea que los detalles de una película, la única diferencia es que las características de la serie son diferentes que las de una película.

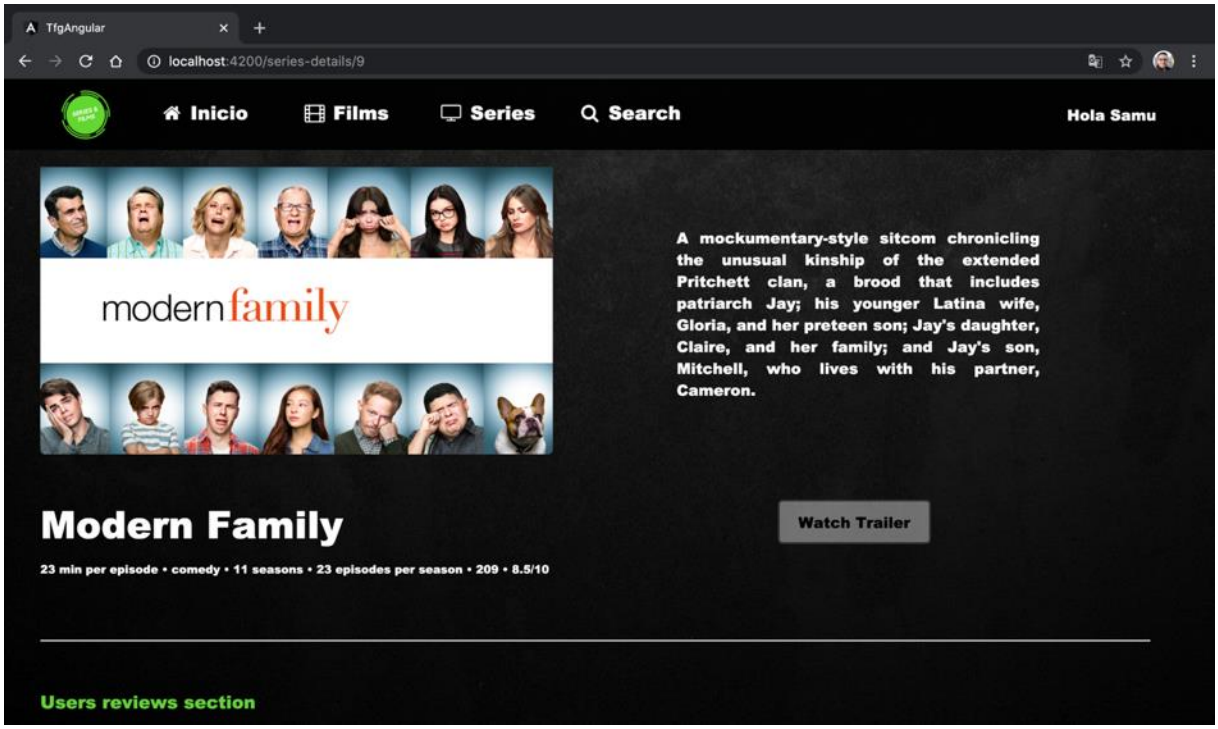

Ilustración 64: Series-details Angular

Users reviews

En este componente podremos ver las reviews del título del que estemos viendo sus detalles. Su apariencia y lo que podemos hacer en él es igual tanto en películas y en series. Como función tenemos la de poder poner nuestras propias opiniones sobre el título y nuestra propia valoración, para que el resto de los usuarios puedan verlas y generar una comunidad en la que los usuarios se interesen tanto como por generar como leer opiniones.

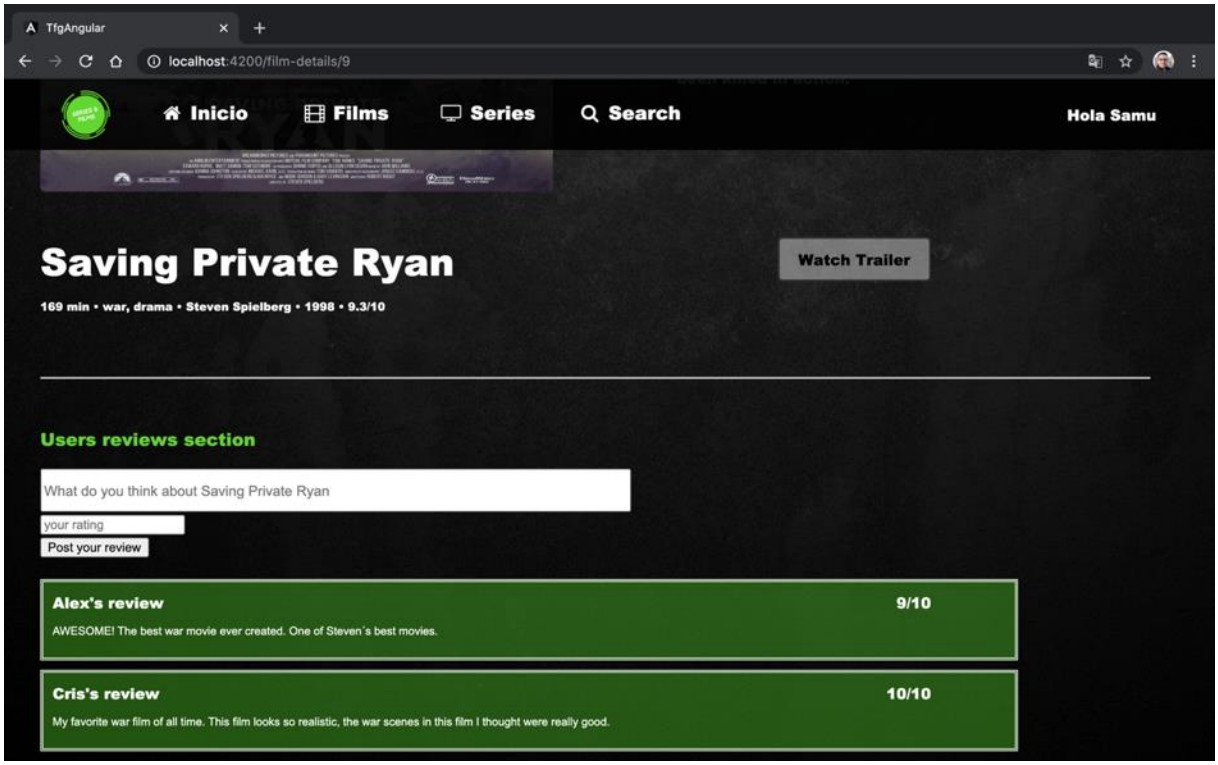

Ilustración 65: Users reviews Angular

Conclusiones

Este proyecto nos ofrece la posibilidad de aprender a fondo sobre la arquitectura de microservicios y su uso en Spring Boot, objetivo que ha requerido de mucho esfuerzo. La síntesis teórica sobre esta arquitectura de la bibliografía existente nos ofrece una visión completa sobre esta, conociendo tanto sus ventajas como desventajas.

Desarrollar este proyecto y su memoria ha sido toda una aventura, en la que ha habido que luchar contra tormentas y mareas, pero el resultado ha sido una victoria muy clara. Hemos conseguido interiorizar los conceptos más importantes de la arquitectura de microservicios, nos hemos familiarizado con Spring Boot y sus diferentes partes y hemos conseguido un diseño de aplicación que es atractivo y útil para entender cómo funciona la arquitectura subyacente existente.

Como partes más complejas del desarrollo podemos mencionar dos como las más complejas. En primer lugar y como tarea más complicada de todas, conseguir interiorizar los conceptos teóricos de la arquitectura y tecnologías con las que hemos desarrollado la aplicación. Sobretodo en esta parte, hemos de destacar el papel de Spring, ya que es un ecosistema inmenso y todas sus partes ofrecen características muy potentes para el desarrollo de aplicaciones, pero comprender qué se está haciendo en cada momento es una tarea terriblemente complicada, pero muy satisfactoria.

La segunda tarea más complicada ha sido la securización del servicio frontera, ya que familiarizarse con Spring Security no es nada fácil, dado que la seguridad es un tema de tanta importancia que podría desarrollarse un trabajo entero únicamente sobre este aspecto del ecosistema de Spring.

El resto del desarrollo ha sido complejo, pero no ha habido demasiados problemas y ha sido un proceso bastante ameno y entretenido, sobretodo el paso al desarrollo de la aplicación Front-End, porque se hacía material todo el esfuerzo en construir un buen Back-End.

En un tono más personal, este proyecto me ha dado la oportunidad de enfrentarme a un desarrollo Full Stack individualmente por primera vez en la vida, y estoy muy satisfecho con el resultado y todos los conocimientos adquiridos por haber elegido este reto. Una experiencia muy curiosa es que gracias a este proyecto (o como efecto secundario de la carga de trabajo) cuando he visto Netflix o Disney + en mis descansos, me he fijado en los posibles servicios que podían estar funcionando detrás de estas aplicaciones web y que yo no tenía pensado desarrollar. Un ejemplo de esto sería un posible servicio (o varios) de los que disponen estas aplicaciones para mostrar el contenido en función de las preferencias personales del usuario.

Trabajos Futuros

Definir las Historias de Usuario fue un proceso complicado, ya que se tenían que fijar unos objetivos muy claros y ceñirse a ellos, dado que se disponía de un tiempo limitado para la consecución de este trabajo.

Algunas características que podrían resultar interesantes para aprender aún más sobre esta arquitectura y Spring Boot, además de las implementadas, podrían ser las siguientes:

La automatización del proceso de despliegue, por medio contenedores como Docker, facilitaría este proceso y sobretodo su uso por otros usuarios en otros equipos

Implementar una conexión con una base de datos para el servicio Zuul y gestionar los usuarios a través de esta, permitiéndonos con esto también dar de alta a otros usuarios,

Implementar un mecanismo de escalado automático, liberando a los servicios de su restricción a un único puerto de nuestro ordenador.

Estos objetivos podrían ser en si mismos trabajos independientes, ya que son procesos complejos, pero su futuro estudio nos daría una mejor visión y experiencia en el desarrollo de arquitecturas basadas en microservicios.

Bibliografía

1. Bruce, M. and Pereira, P., n.d. *Microservices In Action*. 1st ed. New York: Manning Publications.
2. Carnell, J., 2017. *Spring Microservices In Action*. 1st ed. New York: Manning Publications.
3. Christudas, B., 2019. *Practical Microservices Architectural Patterns*. 1st ed. Berkeley: Apress.
4. Finnigan, K., 2019. *Enterprise Java Microservices*. 1st ed. New York: Manning Publications.
5. Macero, M., 2017. *Learn Microservices With Spring Boot*. 1st ed. New York: Apress. DESARROLLO APP
6. Rajput, D. and Rajesh R V, 2018. *Building Microservices With Spring*. 1st ed. Birmingham: Packt Publishing.
7. Richardson, C., 2018. *Microservices Patterns*. 1st ed. New York: Manning Publications.
8. M. Villamizar *et al*., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference (10CCC)*, Bogota, 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.
9. Fernandez D. Arquitecturas basadas en microservicios: Spring Cloud Netflix Eureka - BI Geek Blog [Internet]. BI Geek Blog. 2020 [cited September 2020]. Available from: https://blog.bi-geek.com/arquitecturas-spring-cloud-netflix-eureka/
10. What is Spring Framework: Dependency Injection in Java [Internet]. Marcobehler.com. 2020 [cited September 2020]. Available from: https://www.marcobehler.com/guides/spring-framework
11. Netflix/eureka [Internet]. GitHub. 2020 [cited September 2020]. Available from: https://github.com/Netflix/eureka
12. Netflix/zuul [Internet]. GitHub. 2020 [cited September 2020]. Available from: https://github.com/Netflix/zuul
13. Santana I. Arquitecturas basadas en Microservicios: Spring Cloud Netflix Zuul - BI Geek Blog [Internet]. BI Geek Blog. 2020 [cited September 2020]. Available from: https://blog.bi-geek.com/arquitecturas-basadas-en-microservicios-spring-cloud-netflix-zuul/
14. User Stories | Examples and Template | Atlassian [Internet]. Atlassian. 2020 [cited September 2020]. Available from: https://www.atlassian.com/agile/project-management/user-stories
15. Quijano J. Historias de usuario, una forma natural de análisis funcional [Internet]. Genbeta.com. 2020 [cited September 2020]. Available from: https://www.genbeta.com/desarrollo/historias-de-usuario-una-forma-natural-de-analisis-funcional
16. Top 7 Programming Languages Of 2020 - Coding Dojo Blog [Internet]. Coding Dojo Blog. 2020 [cited September 2020]. Available from: https://www.codingdojo.com/blog/top-7-programming-languages-of-2020
17. Eclipse Foundation I. The Community for Open Innovation and Collaboration | The Eclipse Foundation [Internet]. Eclipse.org. 2020 [cited September 2020]. Available from: https://www.eclipse.org/

18. TOP IDE Top Integrated Development Environment index [Internet]. Pypl.github.io. 2020 [cited September 2020]. Available from: https://pypl.github.io/IDE.html
19. Code V. Visual Studio Code - Code Editing. Redefined [Internet]. Code.visualstudio.com. 2020 [cited September 2020]. Available from: https://code.visualstudio.com/
20. Spring makes Java simple. [Internet]. Spring. 2020 [cited September 2020]. Available from: https://spring.io/
21. Angular [Internet]. Angular.io. 2020 [cited September 2020]. Available from: https://angular.io/
22. Social Network for Programmers and Developers [Internet]. Morioh.com. 2020 [cited September 2020]. Available from: https://morioh.com/p/7687cbec6e12
23. MySQL :: MySQL Workbench [Internet]. Mysql.com. 2020 [cited September 2020]. Available from: https://www.mysql.com/products/workbench/
24. Most Popular Databases In The World [Internet]. C-sharpcorner.com. 2020 [cited September 2020]. Available from: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/
25. Postman [Internet]. Postman.com. 2020 [cited September 2020]. Available from: https://www.postman.com/downloads/
26. Khandelwal A. Spring Boot 2.0—Project Structure and Best Practices (Part 1) [Internet]. Medium. 2020 [cited September 2020]. Available from: https://medium.com/the-resonant-web/spring-boot-2-0-starter-kit-part-1-23ddff0c7da2
27. Khandelwal A. Spring Boot 2.0—Project Structure and Best Practices (Part 2) [Internet]. Medium. 2020 [cited September 2020]. Available from: https://medium.com/the-resonant-web/spring-boot-2-0-project-structure-and-best-practices-part-2-7137bdcba7d3
28. Handle Security in Zuul, with OAuth2 and JWT [Internet]. Baeldung.com. 2020 [cited September 2020]. Available from: https://www.baeldung.com/spring-security-zuul-oauth-jwt

Apéndice A: Glosario

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR