# Borrador TFG - Estudio de la Arquitectura de Microservicios y Spring Boot

1. Introducción
   1. Motivaciones personales
   2. Planteamiento del problema
   3. Objetivos principales
   4. Objetivos adicionales
2. Aplicaciones monolíticas o tradicionales
   1. Características de una aplicación monolítica o tradicional
   2. Riesgos y problemas la arquitectura monolítica o tradicional
3. Aplicaciones basadas en microservicios
   1. Por qué cambiamos la manera de construir aplicaciones
   2. Definición y principios de los microservicios
   3. Características de la arquitectura de una aplicación basada en microservicios
   4. Ventajas frente a la arquitectura monolítica
   5. Transición de una aplicación monolítica a la arquitectura basada en microservicios
   6. El despliegue de microservicios y la nube
   7. Riesgos y dificultades que presentan
   8. Cuando no usar microservicios
4. La importancia de Spring
   1. Spring Boot
   2. Inyección de dependencias
5. Construcción aplicación:
6. Resumen y Conclusión
7. Bibliografía
8. Apéndice. Glosario
9. Introducción
   1. 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.

* 1. 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.

* 1. Objetivos Principales
     1. Comprender qué es un microservicio
     2. Conocer la arquitectura basada en microservicios, sus ventajas y desventajas
     3. Aprender a desarrollar una aplicación en el framework Spring Boot.
  2. Objetivos adicionales
  3. Metodología:

1. Aplicaciones monolíticas o tradicionales
   1. 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 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.

* 1. 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.

1. Aplicaciones basadas en microservicios
   1. 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.
  1. 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.

* 1. 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 extremadamente 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.
  1. 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.
  1. 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.
  1. 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 crucialidad 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 cuál 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 cuál 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.

* 1. 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.

* 1. 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 manejará 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.

1. 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.

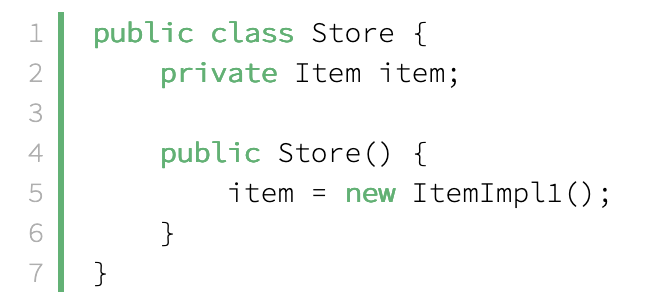
Spring Framework está construido alrededor de la idea de la Inyección de Dependencias (DI), la cuál vamos a explicar de manera detallada.

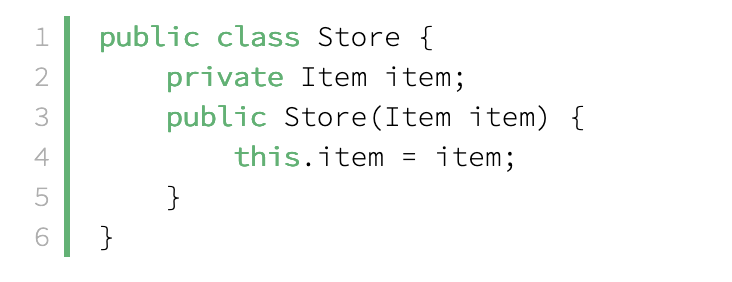
La DI es un patrón que implementa el principio de Inversion of Control (IoC), el cual se basa en que el control de los objetos se delega a un contenedor o un framework. En contraste con la programación tradicional, en la que nuestro código realiza llamadas a una librería, IoC permite al framework tomar 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, crearíamos un objeto de la siguiente manera:



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. 

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); // **(1)**

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 { // **(1)**

@Bean

public DataSource dataSource() { // **(2)**

MysqlDataSource dataSource = new MysqlDataSource();

dataSource.setUser("root");

dataSource.setPassword("s3cr3t");

dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");

return dataSource;

}

@Bean

public UserDao userDao() { // **(3)**

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 cuándo 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, nuestra Bean será inyectada 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 cuál 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.

Captura de pantalla de un celular

Descripción generada automáticamente

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.

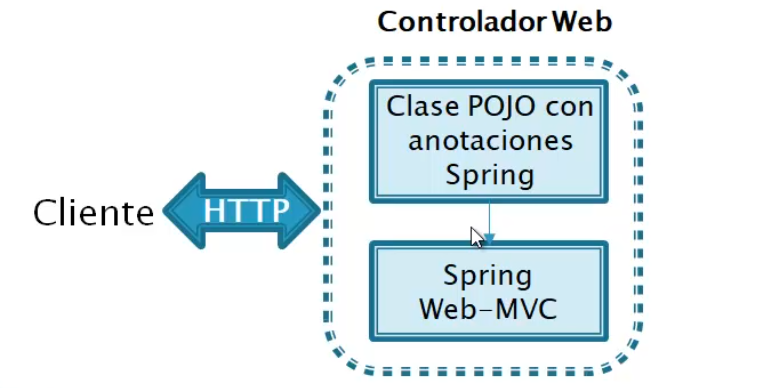
Al igual que Eureka, más adelante se detallará cómo implementar y configurar este servicio frontera.

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 por 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.



* 1. 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:



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: Imagen que contiene captura de pantalla, tabla, pájaro

Descripción generada automáticamente

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.

Captura de pantalla de un celular

Descripción generada automáticamente

## Análisis del problema

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 es 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:  Página principal películas | Prioridad: | Estimado: |
| 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 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Página principal series | Prioridad: | Estimado: |
| 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 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Página principal géneros | Prioridad: | Estimado: |
| Como usuario  quiero una pagina principal que contenga títulos filtrados por género  para poder elegir en función de mis preferencias personales | | |
| Criterios de aceptación:   * Que cuando acceda a la página principal de la aplicación se carguen varias listas que sean de las categorías: COMPLETAR CON GÉNEROS * Que se muestren tanto películas como series. * Que muestre correctamente las películas y series con el género indicado que se encuentran en las BBDD de series y 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Barra de navegación | Prioridad: | Estimado: |
| Como usuario  quiero una barra de navegación en la parte superior  para poder elegir entre 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas catálogo | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas géneros | Prioridad: | Estimado: |
| 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: COMPLETAR GÉNEROS * 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 solape con el resto de las listas. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas características | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas sinopsis | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas portada | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas tráiler | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Películas comentarios | Prioridad: | Estimado: |
| 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. * COMPLETAR SI ES NECESARIO | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series catálogo | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series géneros | Prioridad: | Estimado: |
| 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: COMPLETAR GÉNEROS * 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series características | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series sinopsis | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series portada | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series tráiler | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Series comentarios | Prioridad: | Estimado: |
| 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. * COMPLETAR SI ES NECESARIO | | |

|  |  |  |
| --- | --- | --- |
| Título:  Búsqueda híbrida | Prioridad: | Estimado: |
| 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. | | |

|  |  |  |
| --- | --- | --- |
| Título:  Usuario registrado | Prioridad: | Estimado: |
| Como usuario  quiero  para poder | | |
| Criterios de aceptación: | | |

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: