@MTribor

MICROSERVICIOS

MARCO BORREGUERO TRIGUEROS

I.E.S RAMÓN DEL VALLE INCLÁN

2ºDAM

PROYECTO DE DESARROLLO MULTIPLATAFORMA

Índice

Introducción a la Investigación

Cuando parece que las tecnologías informáticas están tocando techo, siempre aparecen nuevas ideas que no hacen que revolucionar la industria. Y, como no, ese ha sido el caso de los llamados microservicios.

Los servicios web llevan algún tiempo dando de sí. Adaptándose a las nuevas necesidades que el avance tecnológico ha ido exigiendo. Con lenguajes como Java, y, con grandes pilares como el framework Spring, el desarrollo de APIs se ha transformado como algo cotidiano para cualquier desarrollador Software.

Por supuesto, coexisten en el mercado varios protocolos para estos servicios web, como RESTful o SOAP, los cuales se han mantenido como referencia.

Pero, como toda tecnología, acaba por tener sus limitaciones. Por eso no hace tanto tiempo algunas grandes ideas que pretendían ofrecer servicios universales a través de una de las mayores revoluciones tecnológicas en estos tiempos, el *cloud*, vinieron a demostrar que el concepto de servicio utilizado hasta ahora podía tener un nuevo giro.

Y aquí es donde nace este nuevo concepto: los microservicios.

Este concepto nace de la necesidad de integrar los distintos lenguajes, frameworks, protocolos de comunicación, y en general, tecnologías que coexisten en el mercado actual. Con aplicaciones desarrolladas en Java, .Net, Angular JS, Node.js, etc.

Es difícil asegurar una comunicación correcta entre todas estas tecnologías, por lo que es necesario construir un marco de trabajo común que intercepte estas comunicaciones y las transforme en un lenguaje común que sea posible su traducción.

Por lo tanto **¿cómo podemos definir la arquitectura de microservicios?**

> *Se trata de una arquitectura que propone descomponer los distintos componentes que forman una aplicación hasta transformarlos en módulos encargados de realizar una única función con una serie de interfaces bien definidas. Estos módulos han de ser capaces de operar por sí mismos y estás totalmente abstraídos del resto del entorno de la aplicación.*

Objetivos a desarrollar

- **Arquitectura de Microservicios:**
  - Conceptos clave
  - Service Discovery
  - Gateway
  - Server Config
  - Ventajas
  - Inconvenientes
- **Api REST**
- **Docker**
- **Spring Boot y Spring Cloud**

Todo ello será estudiado e investigado a través del desarrollo de una aplicación basada en Microservicios que se tomará como ejemplo práctico y a través de la cual se intentaran entender los conceptos clave de esta arquitectura y su posición actual y a futuro en el mundo Software.

Ejemplo práctico: Tareas de desarrollo

- **Definir los conceptos de nuestro proyecto:**
  - Problema: Introducción al problema y reglas de negocio
  - Modelo: Dominio y entidades
  - Servicios: Tecnologías y organización
- **Desarrollo**
  - Desarrollo de microservicios basados en Api REST
  - Arquitectura de microservicios basada en Spring Cloud
  - Despliegue: Imágenes Docker y Docker-compose

Ejemplo práctico: APP Control de clase

Introducción

Cuando pensamos en aplicaciones que puedan llevar a cabo tareas de organización, evaluación, comunicación o envío de material educativo, entre otras. Posiblemente, se nos vengan a la cabeza una serie de aplicaciones de escritorio que deberemos utilizar de forma individual e independiente. Pero, en la actualidad, ¿seríamos capaces de centralizar todas estas aplicaciones para que funcionen de forma conjunta? ¿Es posible crear un entorno tecnológico capaz de integrar todas las herramientas necesarias para administrar todos los datos?

Estas dudas, se basan en el concepto común de aplicación, ya que éste hace referencia a la arquitectura monolítica en las que se basan todas las aplicaciones de escritorio que utilizamos diariamente.

Entonces, la respuesta es que sí. En la actualidad, con las tecnologías basadas en la nube, las distintas tecnologías de bases de datos, y las aplicaciones web, ya no es una idea inconcebible el hecho de crear sistemas integrados que manejen todos estos datos y sean capaces de llevar a cabo tareas muy distintas de forma orquestada.

Pero ¿qué tipo de aplicaciones? ¿cómo se realiza la integración entre las mismas? ¿bajo qué tecnología? ¿han de integrarse en distintos sistemas operativos?

En este punto, podríamos hablar de los conceptos de elasticidad, alta cohesión y bajo acoplamiento, para los cuales existen muchas arquitecturas software que pueden aportarnos estas propiedades en una aplicación.

En este caso, posiblemente, el uso de una arquitectura basada en microservicios sería la más correcta ya que, de cierta forma, podrían integrarse servicios que ya se encuentren funcionales junto con servicios nuevos totalmente independientes. Además, esta arquitectura permitirá que nuestra aplicación sea **elástica**, de tal forma que permitirá nuevas integraciones de servicios independientes en un futuro.

Gracias a este patrón de diseño, podremos tener acceso a una única aplicación en forma de servicio cloud, que interiormente, se compondrá de distintos servicios, o **módulos,** que podrán trabajar de forma independiente o conjunta según las necesidades del usuario y, sobre todo, de forma anónima entre sí.

Con todo esto, se podrá disponer de una aplicación que conste de distintos servicios totalmente independientes para que podrán ser utilizados en las distintas tareas y en distintos dispositivos a través de otras aplicaciones cliente que los consuman.

Con este supuesto, da comienzo una investigación que tratará de averiguar los entresijos de esta tecnología y su posible aportación positiva a un entorno educativo en el que, cada vez más, es necesario que se integren las tecnologías para un mejor manejo de datos.

Microservicios

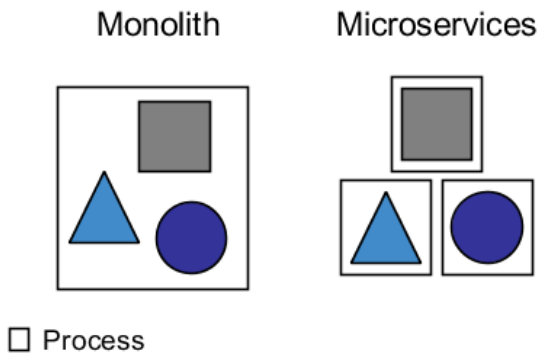
Hasta ahora, los sistemas se han estructurado mediante grandes aplicaciones en forma de monolitos, haciendo que el mantenimiento y la evolución sea demasiado compleja conforme estos proyectos se agrandan. En frente a esta tecnología, tenemos a los microservicios, que consisten en una arquitectura software que implementa servicios mediante la colaboración de otros servicios más pequeños y autónomos.

Cada microservicio se centra en una parte individual del modelo negocio. Así, se consigue la abstracción del resto del sistema de los detalles concretos de la implementación.

Además, esto permite que el despliegue de estos servicios pueda realizarse de forma individual y separada, por lo que el usuario final no tiene que ser conocedor de qué servicios son los que le están ofreciendo los recursos que está pidiendo. Esto, favorece, entre otras cosas, la escalabilidad.

Así pues, podemos enumerar ciertas ventajas de utilizar una arquitectura basada en microservicios para nuestra aplicación:

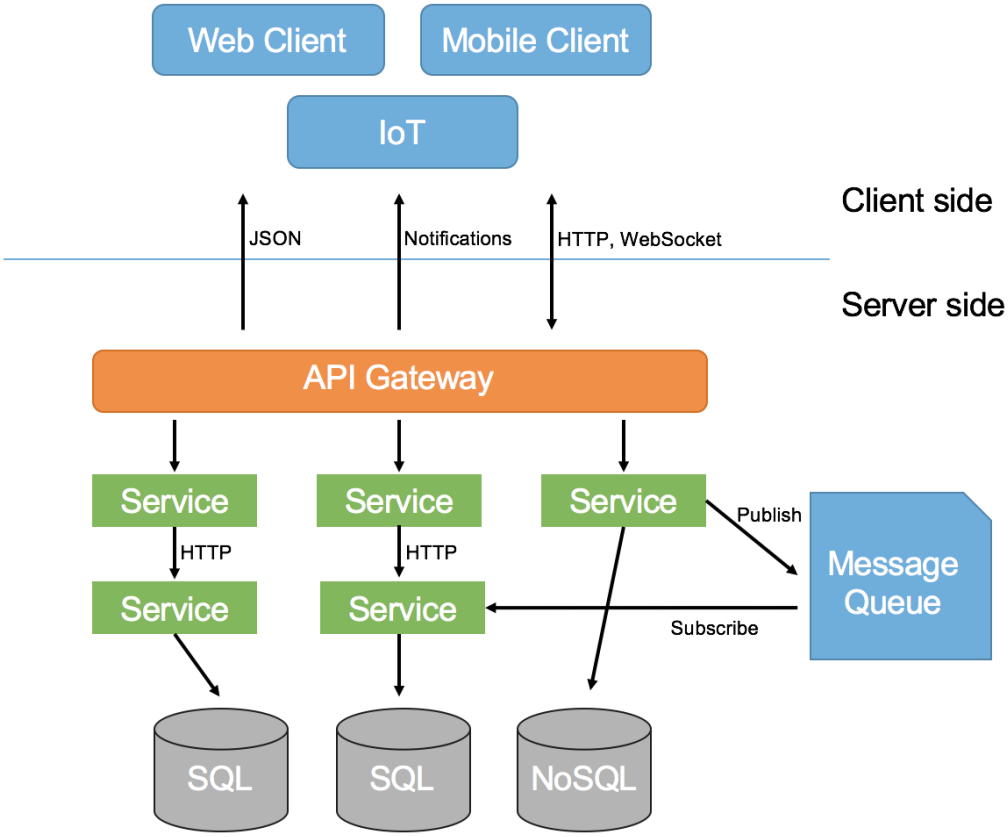
- Facilita la escalabilidad y la hace más eficiente
- La capacidad de prueba de cada microservicio es mayor.
- El mantenimiento se hace más sencillo y eficiente.
- Permite que en un mismo proyecto coexistan distintas tecnologías: Múltiples lenguajes, bases de datos, protocolos, etc.
- Hace mucho más sencillo el desarrollo paralelo de las distintas funcionalidades de la aplicación y, además, permite que este desarrollo sea independiente.
- Permite que los despliegues sean independientes, lo que aumenta la tolerabilidad a fallos graves.

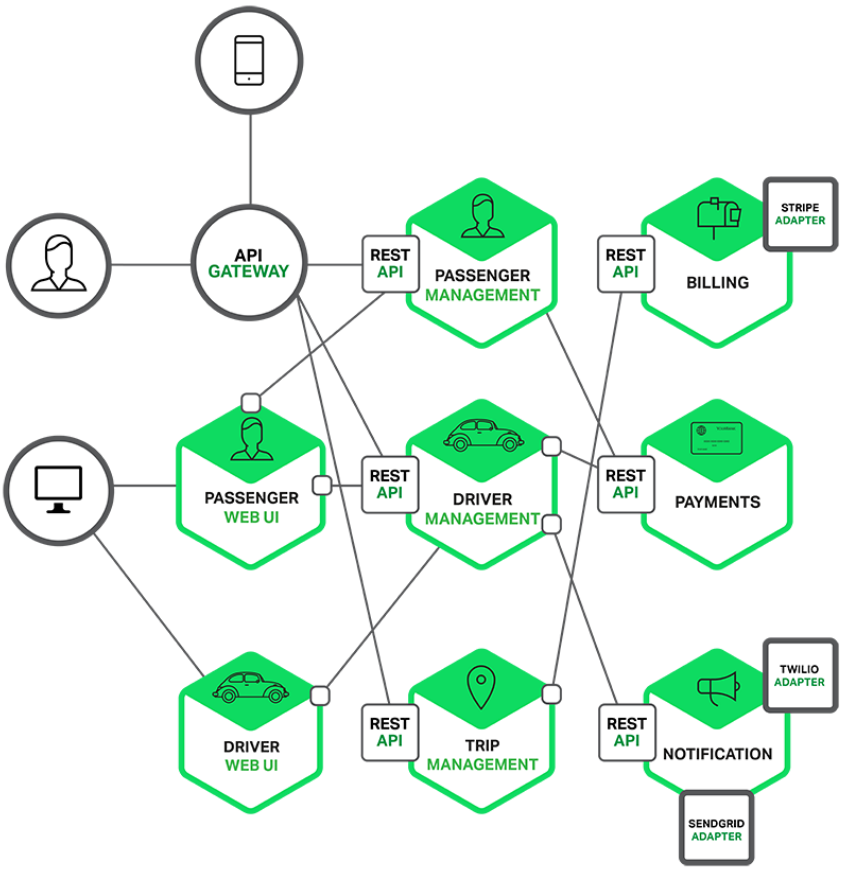


Estructura de la aplicación

La arquitectura basada en microservicios puede tener tantas capas, servicios o componentes, como se requieran. Generalmente, las capas básicas de esta arquitectura son: API Gateway, Service Discovery, Servicios y sus correspondientes bases de datos.

Algunos ejemplos de arquitecturas de aplicaciones basadas en microservicios pueden ser:





Como vemos, una arquitectura de Microservicios se basa en dividir los grandes servicios web utilizados hasta ahora, en pequeños servicios independientes con un dominio más ligero y una lógica con una menor complejidad.

En el caso anterior, por ejemplo, podemos ver como la aplicación posee hasta seis micro-servicios para dar un servicio común a través de la interfaz de aplicación, o API, llamada Gateway en este caso.

Como vemos, estos servicios también pueden respaldarse entre sí, aunque estos se mantengan totalmente independientes. Esto se debe al principio de deslocalización en el que se basan los microservicios.

Por otro lado, podemos ver como no es necesario que todos los servicios mantengan los mismos protocolos de comunicación, es decir, generalmente existirá un mismo protocolo de comunicación, que normalmente será el REST, pero es posible que cada servicio pueda implementar otro u otros protocolos para hacer más flexible la aplicación.

Problema

Se busca mantener un intercambio de información sobre las tareas diarias de las materias, entre los alumnos y el profesor, de una forma ordenada y eficiente.

Es decir, nuestro problema reside en encontrar una forma de realizar un seguimiento centralizado para todas las tareas de una materia. De tal forma que el profesor pueda saber qué alumnos han realizado la tarea, y en caso de no haberla realizado, poder encontrar una breve explicación de por qué.

Por otro lado, el profesor desea conocer qué opinan los alumnos de la dificultad, o el volumen de trabajo, de las tareas que se han pedido. De tal forma que cada alumno pueda valorarla y, además, escribir un breve comentario.

Finalmente, es importante que los alumnos puedan tener acceso a las tareas que se han pedido de forma ordenada cronológicamente. Es decir, de alguna forma, que dichas tareas estén reflejadas en un calendario para que sea fácil para los alumnos, y a la vez para el profesor, organizar dichas tareas en el tiempo.

Modelo

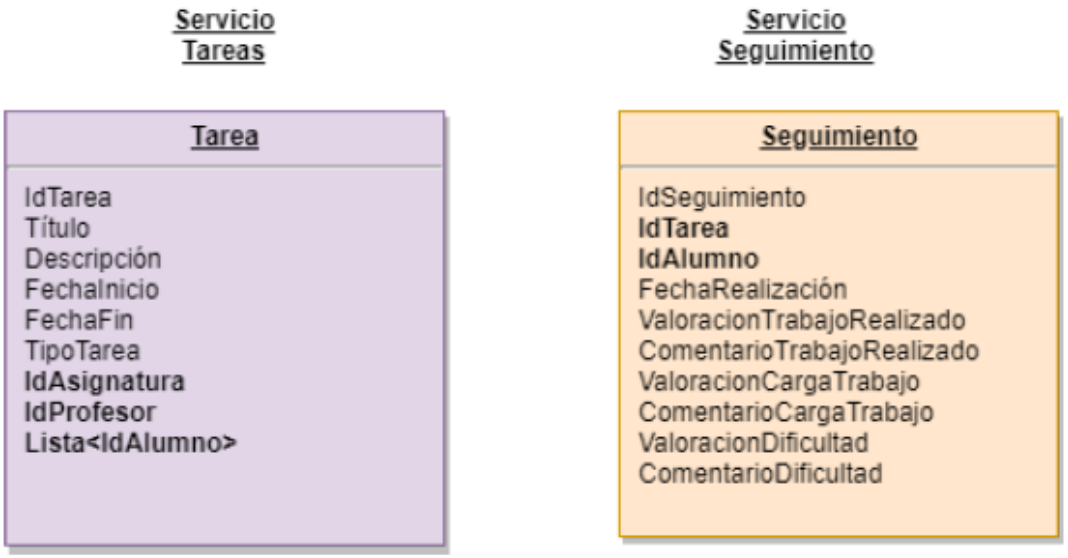
Una vez abordado el problema, se puede construir el modelo para la aplicación. Para ello han de definirse los conceptos con los que se van a trabajar:

Tareas: Son las distintas actividades que se realizan en una materia. Pueden ser de distintos tipos, entre los que se contemplan: Deberes, Exámenes, Talleres, Entregas de trabajos, etc.

Seguimientos: Representan los datos que intercambian los alumnos para cada tarea con el profesor. Corresponden a la valoración del trabajo realizado, la valoración de la dificultad de la tarea y los respectivos comentarios a ambas valoraciones.

Profesores: Son los encargados de administrar las tareas, es decir, son los encargados de crear, modificar y eliminar las tareas. Además, podrán acceder a los comentarios de los alumnos.

Alumnos: Pueden acceder a la información de cada tarea. Además, deberán interactuar con cada tarea para realizar el seguimiento de éstas. Para ello deberán valorar cada tarea y escribir un comentario para cada valoración antes de la fecha de fin de la tarea. Aunque podrán escribir comentarios tras la fecha de fin que, por defecto, valorarán su trabajo realizado como “Fuera de plazo”.



Servicios

Calendario (Tareas)

- **Descripción**: Este servicio soporta las peticiones referidas a las tareas de cada materia en una fecha y horario concreto sobre un calendario y un profesor concreto.
- **Tecnología**: Api Rest, Spring-Boot, Spring-MVC, Lombock, Java 8.
- **Funcionalidad**: Manejo de tareas: Creación, modificación, borrado y lectura de las tareas generadas.
- **Modelo**: Una colección “Tareas” que maneja documentos de tipo “Tarea” que contienen referencias a las materias, alumnos y profesores que la componen.
- **Base de datos**: MongoDB.

Seguimiento (Valoraciones y comentarios)

- **Descripción**: Servicio encargado de manejar las peticiones sobre la realización de las tareas por parte de los alumnos y la valoración de la carga de trabajo y la dificultad de las tareas asignadas.
- **Tecnología**: Api Rest, Spring-Boot, Spring-MVC, Lombock, Java 8.

- **Funcionalidad**: Manejo de controles o seguimientos para las tareas existentes. Creación de un control para una tarea, borrado, modificación y lectura.

- **Modelo**: Una tabla “Seguimientos” que contiene elementos “Seguimiento” los cuales, cada uno, tienen una referencia a cada tarea y cada alumno correspondiente.

- **Base de datos**: PostgreSQL.

API Gateway

- **Descripción**: Servicio encargado de centralizar las llamadas a los demás servicios a través de una URI que hace de entrada de peticiones.

- **Tecnología**: Api Rest, Spring-Boot, Spring-Cloud Gateway sobre Java 8.

- **Funcionalidad**: Se encarga de centralizar las llamadas a la aplicación en una URI principal que redirige las llamadas a los servicios configurados internamente.

Service-Discovery

- **Descripción**: Servicio encargado de registrar las direcciones a los microservicios que componen la aplicación y redireccionar las peticiones hacia estos.

- **Tecnología**: Spring-Boot, Spring-Cloud y Netflix Eureka sobre Java 8

- **Funcionalidad**: Encargado de redirigir las llamadas realizadas a cada servicio a través de una URI genérica a la dirección del servidor en el que se encuentra dicho servicio.

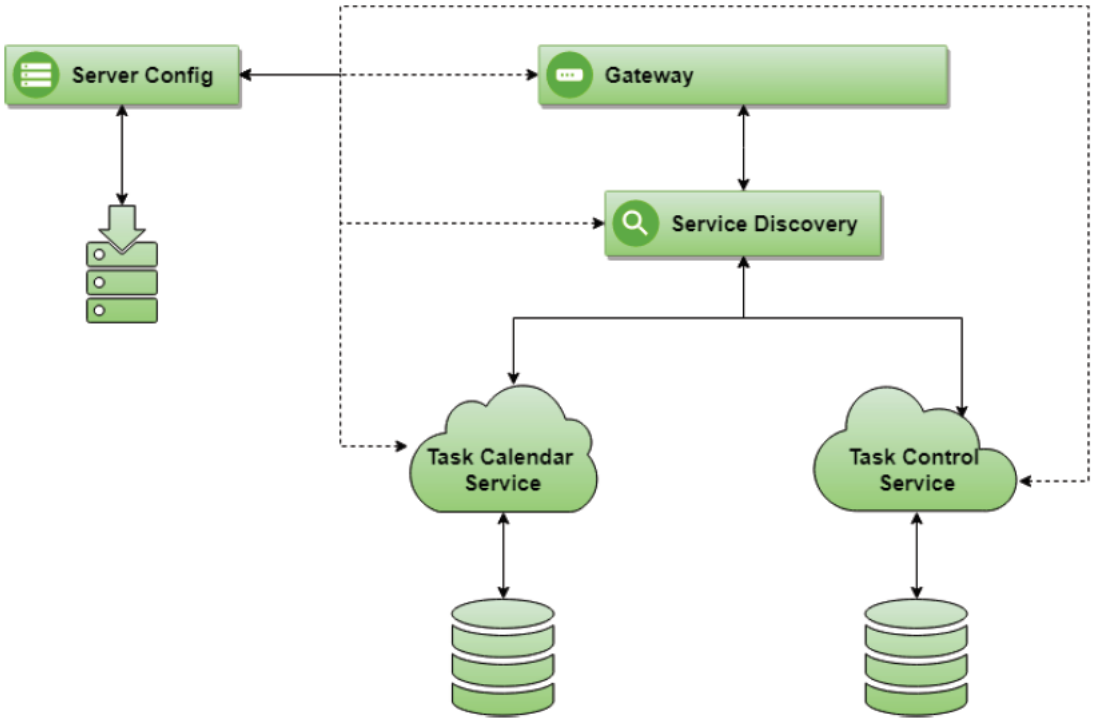
Server Config

- **Descripción**: Este servicio está a la espera de recibir peticiones por parte del resto de servicios, para que se les proporcione la configuración necesaria para su ejecución en el entorno definido. Cuando este servicio recibe estas peticiones, recoge los archivos de configuración para cada servicio (de extensión “.yml” o “.properties”) de un repositorio definido para luego proporcionárselo al servicio que haya realizado la petición.

- **Tecnología**: Spring-Boot y Spring-Cloud: Spring Cloud Config Server sobre Java 8

- **Funcionalidad**: Busca sobre un repositorio los archivos de configuración de cada servicio que los solicite.

En nuestro caso, la arquitectura de nuestra aplicación será la siguiente:



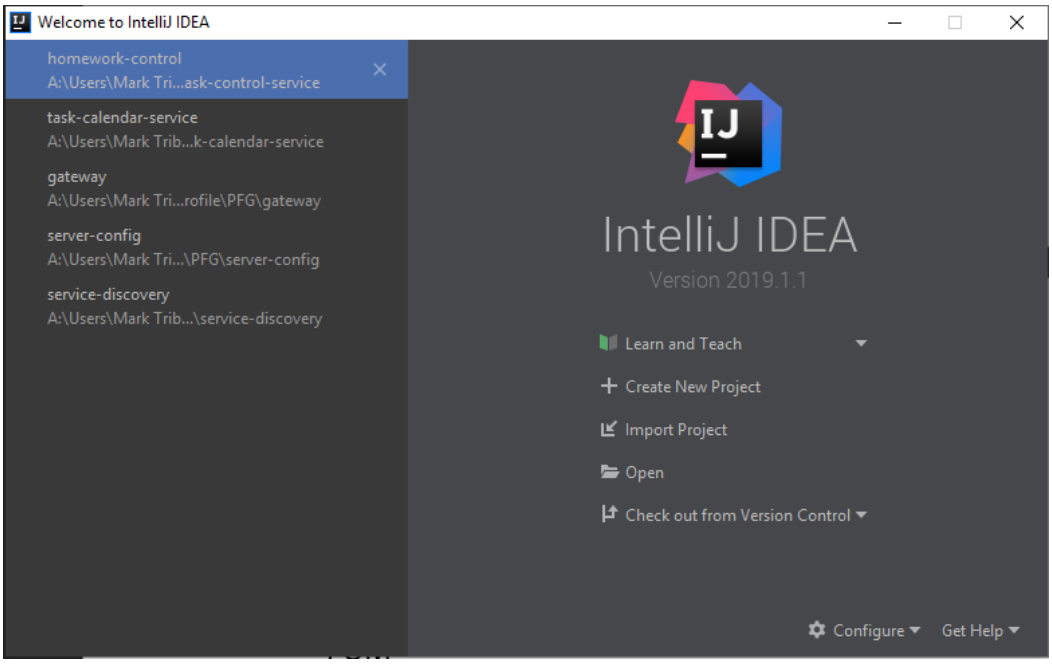
Desarrollo

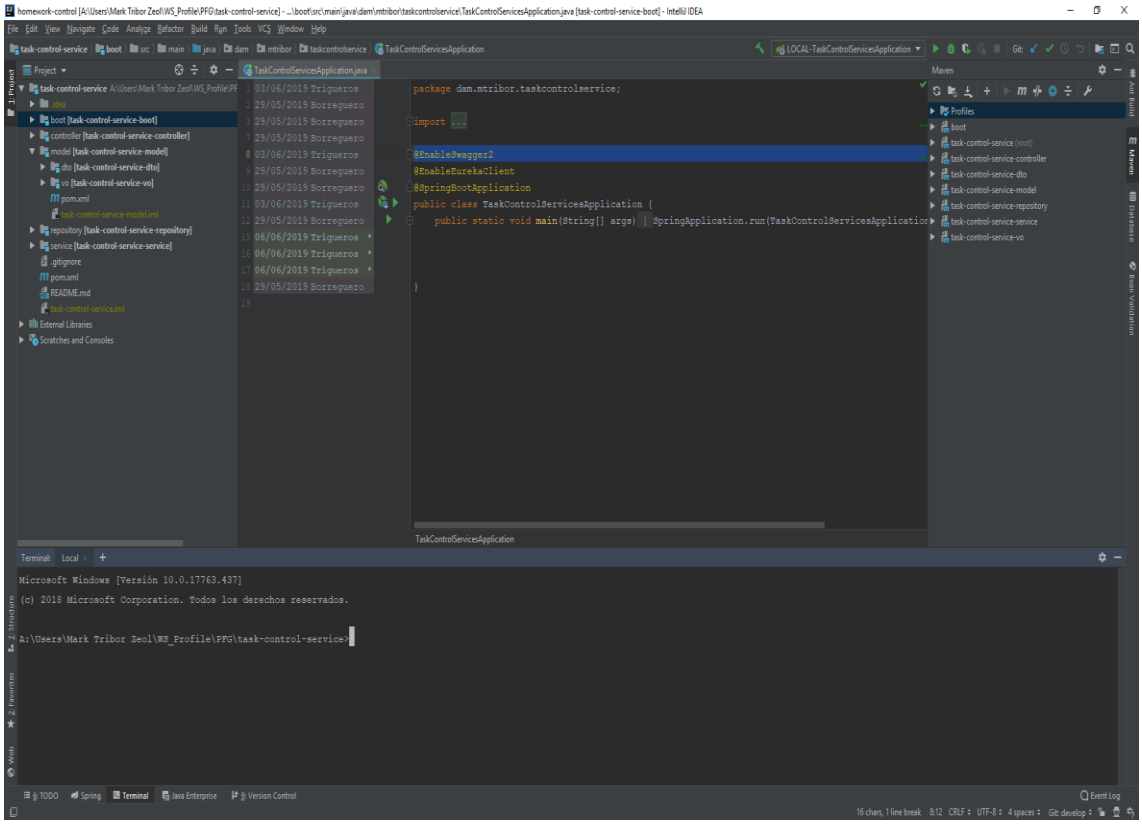
Entorno de desarrollo

IntelliJ IDEA- JetBrains

Para el desarrollo de esta aplicación se utilizará el reconocido IDE IntelliJ. Esto se debe a que posiblemente se trata del IDE que, actualmente, incorpora más y mejores herramientas para el desarrollo de aplicaciones web.

En este caso, además, se trata de un IDE con total compatibilidad con el desarrollo de microservicios.





Web ➞ https://www.jetbrains.com/idea/

Maven – Apache

Maven nace como una herramienta dedicada a facilitar la compilación y generación de ejecutables para aplicaciones. Así como para facilitar la gestión de dependencias y librerías.

Actualmente, para todo desarrollador, es necesario tener en el desarrollo de sus proyectos un gestor de dependencias y librerías. Esto se debe a que una de las grandes premisas de la programación se basa en la no repetición de código, la generalización, la parametrización y la modularidad.

En este caso se utilizará Maven como gestor de dependencias para nuestro proyecto, ya que es uno de los más potentes del mercado y con una gran versatilidad con respecto al desarrollo de microservicios.

La administración de las dependencias y librerías utilizadas en el proyecto se realiza a través de unos archivos con formato XML y al que nos referiremos como el archivo POM de la aplicación (POM.xml).

Un ejemplo de archivo POM utilizado en el desarrollo de esta aplicación, puede ser este:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <packaging>pom</packaging>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.5.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>


    <modules>
        <module>boot</module>
        <module>model</module>
        <module>service</module>
        <module>controller</module>
        <module>repository</module>
    </modules>


    <groupId>dam.mtribor</groupId>
    <artifactId>task-control-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>task-control-service</name>
    <description>Homework Control service for the Task Control
APP.</description>

    <properties>
        <java.version>1.8</java.version>
```

```
<netflix.eureka.version>2.1.1.RELEASE</netflix.eureka.version>
        <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.retry</groupId>
            <artifactId>spring-retry</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>

        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.9.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.9.2</version>
        </dependency>
    </dependencies>
```

```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
            </plugin>
            <plugin>
                <groupId>io.fabric8</groupId>
                <artifactId>fabric8-maven-plugin</artifactId>
                <version>3.5.41</version>
                <configuration>
                    <generator>
                        <includes>
                            <include>spring-boot</include>
                        </includes>
                        <config>
                            <spring-boot>
                                <registry>eu.gcr.io</registry>
                                <name>task-control-service:%v</name>
                            </spring-boot>
                        </config>
                    </generator>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Web ➞https://maven.apache.org/

Spring Framework – Pivotal

Spring nace como sustituto, o complemento, al modelo EJB de Java referente a la plataforma Java EE. Es decir, fue diseñado con la intención de agilizar el desarrollo de aplicaciones empresariales.

Tras más de una década de evolución y desarrollo hoy en día Spring ofrece una serie de módulos que pueden trabajar, o no, de forma conjunta y que conforman el llamado "Spring Framework".

Entre los módulos más relevantes para este proyecto, encontramos:

- **Spring Boot**: Se trata del módulo que permite y arrancar y configurar de forma rápida y sencilla aplicaciones basadas en Spring.

- ***Spring Cloud***: Da a los desarrolladores herramientas para crear de forma rápida y sencilla aplicaciones distribuidas basadas en entornos en la nube y facilita la configuración de forma centralizada y "cloud-native".
- **Spring Data**: Su propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.

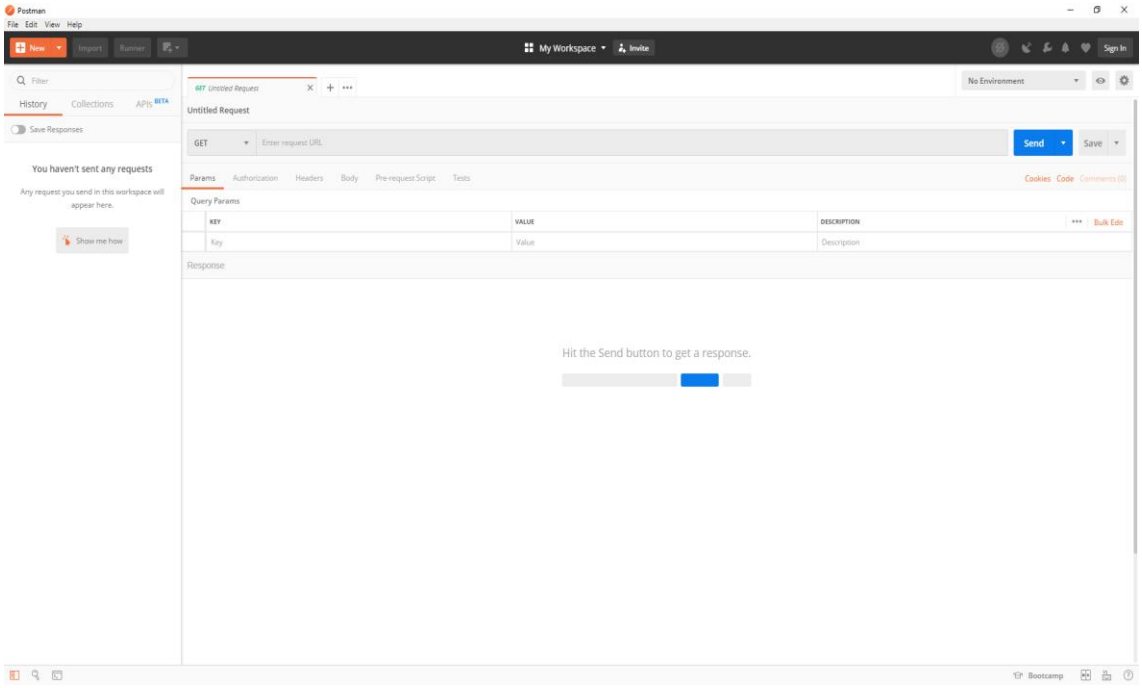
Web ➞ https://spring.io/

Postman - Postdot Technologies

Postman nace como una extensión para el navegador web Google Chrome con la propuesta de hacer más fácil a los desarrolladores realizar pruebas con aplicaciones web realizando peticiones específicas y analizar sus respuestas.

Postman se compone de diferentes herramientas que permiten realizar tareas diferentes con las API REST. Desde la creación de peticiones a APIs, tests de validación del comportamiento de éstas, la posibilidad de crear diferentes entornos de trabajo y parametrizarlos y, además, ofrece la posibilidad de hacer todas estas tareas junto con tu equipo de desarrollo, ofreciendo la posibilidad compartir tus datos con otros compañeros y de exportar los mismos.

Actualmente Postman se presenta como una aplicación de escritorio que permite realizar todas estas tareas de una forma más potente y con la integración de ciertos servicios en la nube.



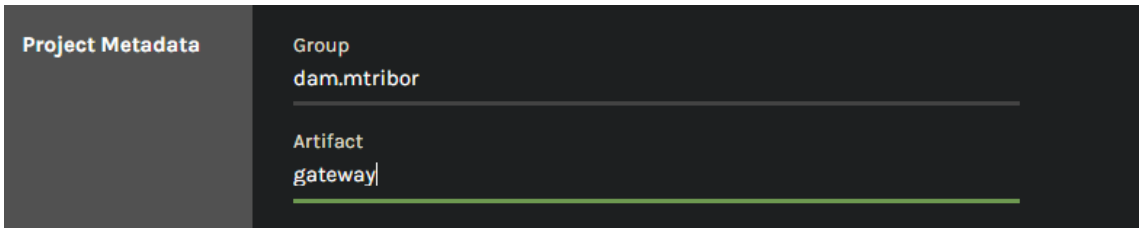
Web ➞ https://www.getpostman.com/

Proyecto Maven

Este proyecto se construirá utilizando el administrador de librerías y dependencias Maven. Por lo tanto, primero, han de configurarse adecuadamente los metadatos de la aplicación como son el Group ID o el Artifact ID de cada componente o módulo, entre otros.

En nuestro caso, el patrón que se utilizará como group-id en todos los proyectos es el siguiente: “dam.mtribor”. Donde “dam” corresponde a “Desarrollo de Aplicaciones Multiplataforma” y “MTribor” al nombre del desarrollador o la organización, empresa o equipo de trabajo.

Por otro lado, el artifact-id hace referencia al identificador que Maven utilizará para resolver las dependencias entre módulos. Es decir, cada módulo que se desarrolle contendrá un archivo “pom.xml” en el que constará, entre otros datos, su id de artefacto para que los demás módulos puedan acceder a sus paquetes. Generalmente, este id contendrá el nombre del módulo que se está desarrollando, como por ejemplo “model”, “repository”, “gateway”, etc.



Además, para cada módulo, será necesario crear el archivo "pom.xml" para las dependencias entre módulos y con librerías externas que se necesiten. Para esto, siempre es necesario tener en cuenta que estas dependencias se traspasan de padre a hijo, pero no al revés.

Para conocer qué dependencias necesita nuestro proyecto, es posible realizar búsquedas sobre qué tipos de librerías se ofrecen que nos puedan resultar de utilidad. Luego, solo será necesario copiar sus descriptores de Maven y colocarlos en los archivos POM de cada módulo.

Por ejemplo, las dependencias básicas para crear una aplicación Spring Boot las podemos encontrar en la web principal de Spring o en el repositorio de Maven Central. Para crear una aplicación básica con Spring Boot, tan solo necesitaremos la siguiente dependencia en el POM de nuestra aplicación:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Además, será necesario establecer nuestra aplicación como hija de una aplicación Spring Boot. Para ello, solo habrá que añadir las siguientes líneas:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.5.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

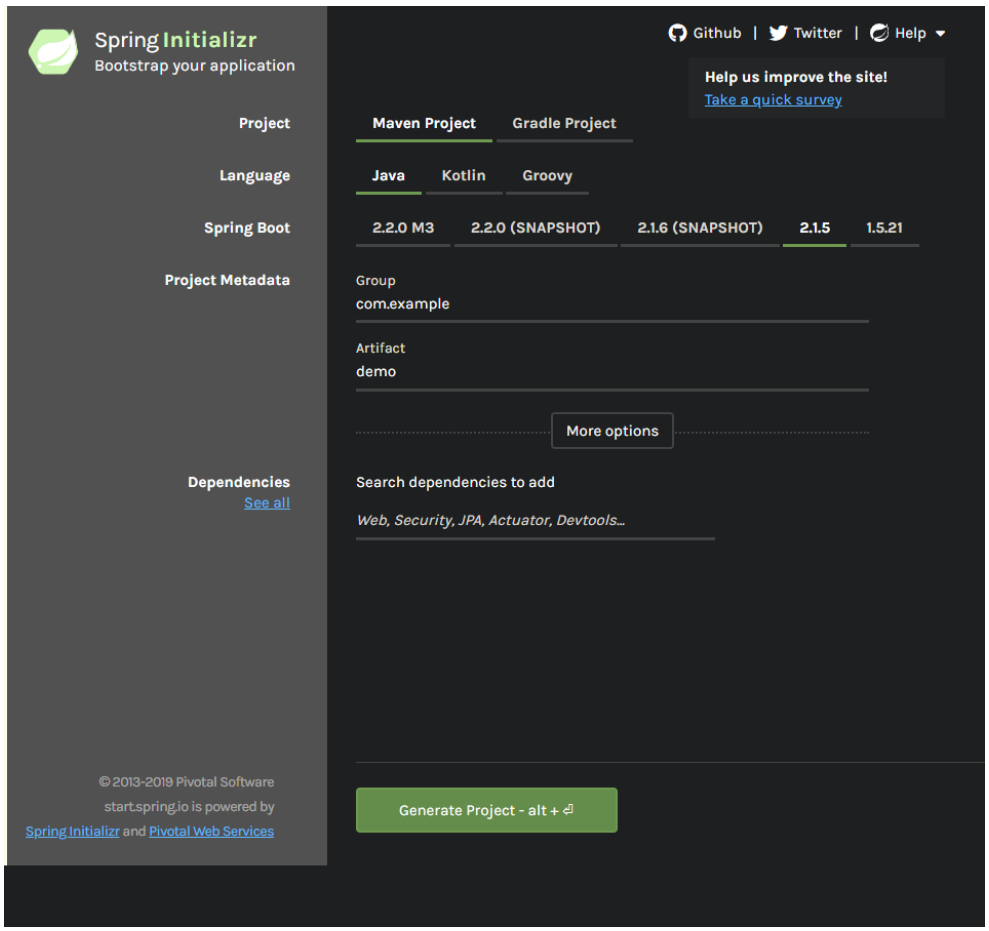
Por último, será necesario establecer el plugin de Spring Boot a través del cual de compilará nuestro proyecto y el cual permitirá a Spring analizar el proyecto en busca de las dependencias y los java-beans necesarios para la ejecución de su contexto:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Todas las librerías disponibles las podemos encontrar en la web oficial de Maven: https://mvnrepository.com/. Además, existen otras dos webs muy útiles para nuestro caso:

- **Search Maven:** Una web creada por la comunidad que facilita la búsqueda de librerías actualizadas y funcionales en el repositorio de Maven - https://search.maven.org/

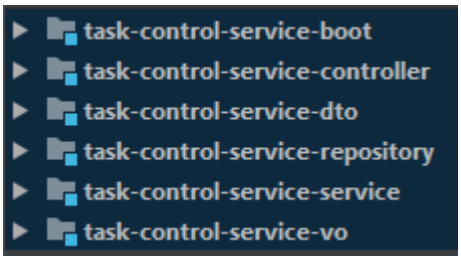
- **Spring Initializr:** Se trata de una herramienta oficial de los desarrolladores del famework Spring, a través de la cual se puede inicializar un proyecto sencillo con la mayoría de las librerías esenciales para el uso del framework -https://start.spring.io/



Proyecto Multimódulo

Desde siempre se ha hecho uso de librerías, externas o internas, para ser reutilizadas durante el desarrollo software. Es decir, reutilizar software ya existente y de esta forma generalizar ciertas funcionalidades comunes.

En este caso, si nos basamos en el servicio desarrollado para el Control de las tareas de clase para cada alumno, dicho servicio se encuentra dividido en seis módulos distintos:



- Boot ➞Es el módulo donde encontraremos la clase de ejecución principal o “main”. Contendrá cierta configuración de importancia y será el módulo hijo del paquete principal.

- Controller ➞Define el manejo de las peticiones HTTP recibidas. En este caso, se definirá con la anotación @RestController y se deberán definir las rutas para las uris de acceso, así como el método de cabecera que deberá llevar la petición (GET, POST, PUT, PATCH, etc.)

- Service ➞ Se trata de la capa donde reside la lógica de nuestra aplicación, es decir, será el módulo encargado de interactuar con el modelo de datos, el repositorio y los datos y peticiones recibidos desde la capa controlador.

- Repository ➞En este módulo se definirá el manejo de datos. Estos datos podrán ser recibidos a través de interfaces de conexión con bases de datos, o datos externos recibidos a través de APIs externas o propias.

- Model ➞ Se trata de la capa “dominio” de nuestra aplicación, y define las entidades y objetos con los que nuestra aplicación trabajará.

    - DTO➞Se trata de un módulo opcional que será hijo del módulo “Model” y contendrá sólo los objetos DTO para que puedan ser integrados por otras aplicaciones gracias a que son un módulo.

    - VO ➞ Al igual que el anterior, se trata de un módulo opcional que trata de modularizar la capa de dominio de la aplicación.

De esta forma, a través del gestor Maven, nos encontramos con la posibilidad de construir proyectos basados en módulos, los cuales son gestionados como dependencias Maven a través de los archivos POM que cada uno contiene.

```
<modules>
    <module>boot</module>
    <module>model</module>
    <module>service</module>
    <module>controller</module>
    <module>repository</module>
</modules>
```

Spring Boot

Spring Boot nos ofrece crear aplicaciones aplicaciones Spring sin tener que realizar el tedioso proceso de configuración. No es necesario hacer ningún fichero de configuración.

Por ejemplo, para la creación de una aplicación web, Spring Boot, automáticamente nos proporcionará un servidor Tomcat embebido el cual estará configurado y conectado junto con un servlet propio de Spring. Además, por supuesto, toda la configuración por defecto será personalizable.

Podemos resumir el éxito de Spring Boot en las siguientes características:

- **Configuración**: Spring Boot cuenta con un complejo sistema de contexto de aplicación, que escanea y analiza la aplicación dividiéndola por componentes en tiempo de compilación y ejecución. Spring Boot es capaz de autoconfigurar todos los aspectos de una aplicación y permitir ejecutarla sin necesidad de definir absolutamente nada.
- **Gestión de dependencias:** Tan sólo es necesario indicar el tipo de proyecto que se desarrollará Spring Boot será el encargado de resolver las dependencias necesarias entre módulos y componentes para que la aplicación funcione.
- **Despliegue**: Spring Boot es capaz de ejecutarse tanto como aplicación Stand-alone como aplicación web mediante su servidor web integrado.
- **Modularidad**: Spring Boot se basa en una estructura modular independiente que permite que la comunidad diseñe y proponga nuevos módulos para integrarlos en su desarrollo.

En nuestro caso, al tratarse de una aplicación multimodular y basada en Spring Boot, han de señalarse adecuadamente las dependencias entre módulos. Esto es: a parte de la inyección de dependencias entre módulos, han de añadirse dichas dependencias al POM de cada módulo, ya que cada uno de éstos actúa como si fuera un proyecto independiente y, por lo tanto, los otros módulos se añaden a él como si fueran una librería.

En el caso de realizar un proyecto multimodular utilizando Spring Boot han de tenerse en cuenta las siguientes advertencias que se han detectado:

- Se deben optimizar las dependencias de tal forma que no existan dependencias cíclicas.
- Es recomendable crear un módulo “boot” que parta del módulo padre de la aplicación y el cual contenga la clase principal de ejecución para el contexto de Spring. Este contexto será el encargado de escanear nuestro proyecto en busca de todas las clases necesarias en el orden necesario junto a la configuración requerida para que la aplicación Spring se ejecute correctamente.

```
@EnableSwagger2
@EnableEurekaClient
@SpringBootApplication
public class TaskControlServicesApplication {
    public static void main(String[] args) {
SpringApplication.run(TaskControlServicesApplication.class, args);
}}
```

- Es necesario colocar la descripción del plugin principal de Spring en el POM del módulo boot antes mencionado, junto a la dependencia con el módulo controlador (@RestController) de la aplicación.

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>task-control-service-controller</artifactId>
    <version>${project.version}</version>
    <scope>compile</scope>
</dependency>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Mediante Spring Boot tenemos la posibilidad de crear aplicaciones completas ejecutables en pequeños entornos para los cuales ya no es necesario tener un servidor de aplicaciones completo junto con una gran cantidad de recursos.

Spring boot, además, tiene complementos muy interesantes como Spring Cloud con los cuales se nos permite crear aplicaciones “Cloud Native”, o lo que es lo mismo, aplicaciones diseñadas desde el inicio para ser ejecutadas en la nube.

Estas ventajas, son clave para la creación de microservicios, ya que hacen que no sea necesaria la instalación de servidores, así como la necesidad de realizar complejas configuraciones.

Spring Boot - API Rest

Llamamos **API** al conjunto de funciones y procedimientos que realizan distintas funciones con el fin de ser utilizadas por otro software. Este concepto se encuentra incluido en el concepto software de librería.

Las siglas API vienen del inglés Application Programming Interface, que en español sería Interfaz de Programación de Aplicaciones. Esto se debe a que una API nos permite implementar las funciones y procedimientos que engloba en nuestro proyecto sin la necesidad de programarlas de nuevo. En términos de programación, es una capa de abstracción.

Por otro lado, **API REST** se define como una librería que se basa totalmente en el estándar de comunicación HTTP. Dicho de otra forma, una API REST es un servicio que nos aporta funciones heredadas a través de un servicio web que no es nuestro, dentro de una aplicación propia, de manera segura.

REST, REpresentational State Transfer, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP. Por ello, al ser un estándar de red, requiere de ciertas restricciones que permitan la correcta comunicación entre usuarios. Estas restricciones son:

- **Conexión cliente-servidor libre**; El cliente no necesita saber los detalles de la implementación del server, y este tampoco debe preocuparse por cómo se usan los datos que envía.
- **Cada petición enviada al servidor es independiente.**
- Compatibilidad con un sistema de **almacenamiento en caché.**
- **Cada recurso del servicio REST debe tener una única dirección**, manteniendo una interfaz genérica.
- Permite utilizar **diferentes capas para la implementación del servidor**.

En nuestro caso, los servicios desarrollados siguen este patrón, ya que a día de hoy, REST y JSON, son la combinación más utilizada para el desarrollo de APIs, y la que un mayor apoyo de parte de la comunidad poseen.

Con esto, además, Spring Framework posee toda una serie de librerías, módulos y funcionalidades a disposición de los desarrolladores para crear nuevas APIs REST. Para nosotros, algunas de estas características son:

- La anotación **@RestController**, que indica que la clase Java que la contenga será la encargada de manejar las peticiones HTTP que nuestra aplicación recibirá, y que, además, nuestra aplicación ofrecerá respuestas acordes a las restricciones REST. En nuestro caso, como se ha indicado anteriormente, se oferecerán respuestas JSON.

- Además, Spring nos ofrece una serie de anotaciones para indicar al controlador REST qué métodos Java manejaran qué peticiones HTTP. Estas peticiones HTTP, pueden ser desde GET, POST, PUT PATCH, OPTIONS, DELETE, etc. Por lo tanto, Spring nos ofrece las siguientes anotaciones para cada operación: **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**, etc.

- Es recomendable, que los métodos de la clase "Controller" devuelvan objetos del tipo **ResponseEntity<T>**, donde "T" hace referencia al tipo de objeto que será transformado a JSON a través del traductor interno de Spring, llamado JackSon. Estos objetos, además de incluir el objeto Java a traducir como respuesta, nos permiten personalizar las cabeceras HTTP para dar unas respuestas más complejas y completas.

Como ya hemos dicho, REST nos permite implementar los métodos HTTP y es fácil observar, que existen similitudes entre estos y las operaciones de una interfaz CRUD. Esto se debe a que realmente, una petición GET estará asociada a un query SELECT, una petición POST estará asociada a un query INSERT, una petición PUT a un UPDATE y así sucesivamente. Esto no tiene por qué ser siempre así, ya que por ejemplo, podría darse el caso de que una API consulte información de los recursos o procese ciertos datos para dar una respuesta sencilla. Esto dependerá de las necesidades del proyecto, pero en general funciona de la manera antes mencionada.

Para nuestro desarrollo, en el servicio "Task Control", podemos ver cómo queda definido una interfaz API y su implementación "Controller":

```
@RequestMapping(EndPointUris.CONTROLS)
public interface TaskControlApi {

    @PostMapping
    ResponseEntity<ControlDTO> create(@Valid @RequestBody final
ControlDTO controlDTO);

    @GetMapping
    ResponseEntity<List<ControlDTO>> getAll(final
ControlFilterCriteriaDTO search);

    @GetMapping(EndPointUris.CONTROL)
    ResponseEntity<ControlDTO> getById(@PathVariable(value = "id")
final int controlId);

    @DeleteMapping(EndPointUris.CONTROL)
    ResponseEntity<Void> delete(@PathVariable(value = "id") final
```

```
int controlId);

    @PutMapping(EndPointUris.CONTROL)
    ResponseEntity<ControlDTO> update(@PathVariable(value = "id")
final int controlId,
                                      @Valid @RequestBody final
ControlDTO controlDTO);


}
```

En la Interfaz TaskControlAPI, definimos los métodos que manejará nuestro servicio junto con las anotaciones pertinentes, con la intención de que se puedan abordar varias implementaciones si fuera necesario.

```
@RestController
public class TaskControlController implements TaskControlApi {

    @Autowired
    private TaskControlService taskControlService;

    @Override
    public ResponseEntity<ControlDTO> create(@Valid ControlDTO
controlDTO) {
        return
ResponseEntity.ok(taskControlService.create(controlDTO));
    }

    @Override
    public ResponseEntity<List<ControlDTO>>
getAll(ControlFilterCriteriaDTO search) {
        return ResponseEntity.ok(taskControlService.getAll(search));
    }

    public ResponseEntity<ControlDTO> getById(@PathVariable(value =
"id") final int controlId) {
        return
ResponseEntity.ok(taskControlService.getById(controlId));
    }


    @Override
    public ResponseEntity<Void> delete(int controlId) {
        return ResponseEntity.noContent().build();
    }

    @Override
    public ResponseEntity<ControlDTO> update(int controlId, @Valid
ControlDTO controlDTO) {
        return
ResponseEntity.ok(taskControlService.update(controlId, controlDTO));
```

```
    }
}
```

Spring Boot - API Rest: Manejo de Excepciones

Para el control de excepciones de un servicio REST con Spring existen varias opciones.

Una de las más fáciles, aunque sólo puede usarse a partir de Spring 5, consiste en utilizar la excepción aportada por Spring llamada "**ResponseStatusException**".

```
@GetMapping(value = "/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse
response) {
    try {
        Foo resourceById =
RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new
SingleResourceRetrievedEvent(this, response));
        return resourceById;
    }    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
                HttpStatus.NOT_FOUND, "Foo Not Found", exc);    }}
```

Esta excepción la debemos lanzar en el "catch" de un bloque "try-catch" o al controlar algún comportamiento indebido en nuestra aplicación a nivel de controlador. Es útil para realizar prototipos de métodos y ver las posibles excepciones que puede lanzar un método, antes de implementar un manejador global de excepciones.

***AVISO**: Si se controlan las excepciones con un manejador @ControllerAdvice y además lanzando este tipo de excepción, se crean conflictos.*

Otra opción, es utilizar la notación de Spring **@ControllerAdvice** que permite implementar manejadores globales @ExceptionHandler para distintas excepciones:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
        extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value
            = { IllegalArgumentException.class,
IllegalStateException.class})
    protected ResponseEntity<Object> handleConflict(
            RuntimeException ex, WebRequest request) {
        String bodyOfResponse = "This should be application
specific";
        return handleExceptionInternal(ex, bodyOfResponse,
                new HttpHeaders(), HttpStatus.CONFLICT, request);
```

```
    }
}
```

Está pensado para ser utilizado a nivel de controlador de nuestro servicio rest, ya que hace uso de los objetos de tipo ResponseEntity.

***AVISO**: Si no se especifica una excepción para ser controlada en la anotación "@ExceptionHandler(value= {...AQUÍ...})" se producirá un error indicando que no se ha encontrado manejador definido para esa excepción.*

Por último, se suele utilizar una combinación de dos métodos que consiste en la creación de excepciones propias las cuales no será necesario que lancen a través de la anotación "@ResponseStatus(value = HttpStatus.ERROR_TYPE)" un error de tipo HTTP como respuesta, ya que de eso se encargará el manejador global. Dicho manejardor controlará estas excepciones como un error REST y dará una respuesta a través de los ResponseEntity que le indiquemos (con archivos HTML estáticos por ejemplo).

Por lo tanto, debemos crear una clase controladora y utilizar la anotación **@RestControllerAdvice** para definir un manejador de excepciones global que controle estas excepciones.

Modelo

Se sigue el patrón usual para los modelos de las aplicaciones web. Se distinguen dos tipos de objetos diferenciados, que son los Value Objects y los Data Transfer Objects:

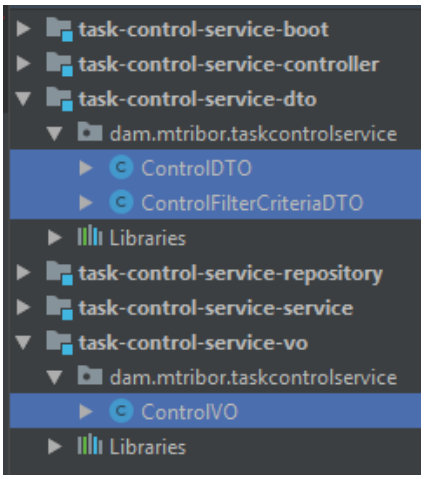
VO (Value Object) → Son los datos que se pueden o se encuentran persistidos en la BD. Es decir, son el tipo de datos que se intercambian con la base de datos y que quizás requieran tener ciertas propiedades específicas para su manejo en la base de datos. Además, en el contexto de Spring, existen ciertas anotaciones que pertenecen al módulo de Spring Boot, llamado Spring Data, y que permiten configurar de forma sencilla este tipo de objetos.

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Entity
public class ControlVO {

    @GeneratedValue
    @Id
    private Integer id;
    private String task_reference;
    private String alum_reference;
    private Date controlDate;
    private Integer workDoneQuantityValuation;
    private String workDoneQuantityComment;
    private Integer workLoadValuation;
    private String workLoadComment;
    private Integer difficultValuation;
```

```
    private String difficultComment;
}
```

DTO (Data Transfer Object) → Son la representación de los datos que se manejan en la aplicación para ser transferidos o recibidos como recursos en las peticiones realizadas hacia la aplicación. Solo son utilizados en el intercambio de datos en las peticiones web.

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ControlDTO {
    private Integer id;
    private String task_reference;
    private String alum_reference;
    @DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX")
    private Date controlDate;
    @NotNull(message = "The evaluation for the quantity of the work
done can't be null!")
    private Integer workDoneQuantityValuation;
    @NotBlank(message = "The comment for the quantity of the work
done can't be null!")
    private String workDoneQuantityComment;
    @NotNull(message = "The evaluation for the workload can't be
null!")
    private Integer workLoadValuation;
    @NotBlank(message = "The comment for the workload can't be
null!")
    private String workLoadComment;
    @NotNull(message = "The evaluation for the difficult can't be
null!")
    private Integer difficultValuation;
    @NotBlank(message = "The comment for the difficult can't be
null!")
    private String difficultComment;
}
```

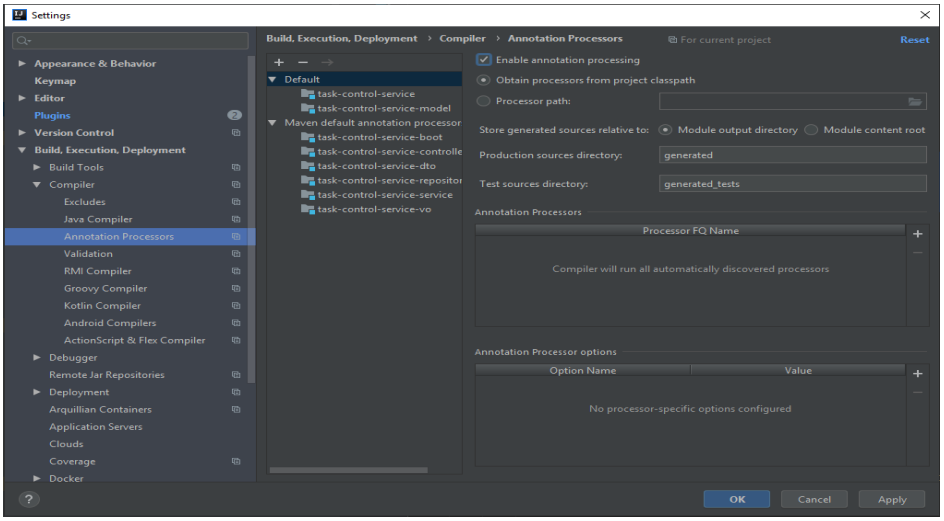


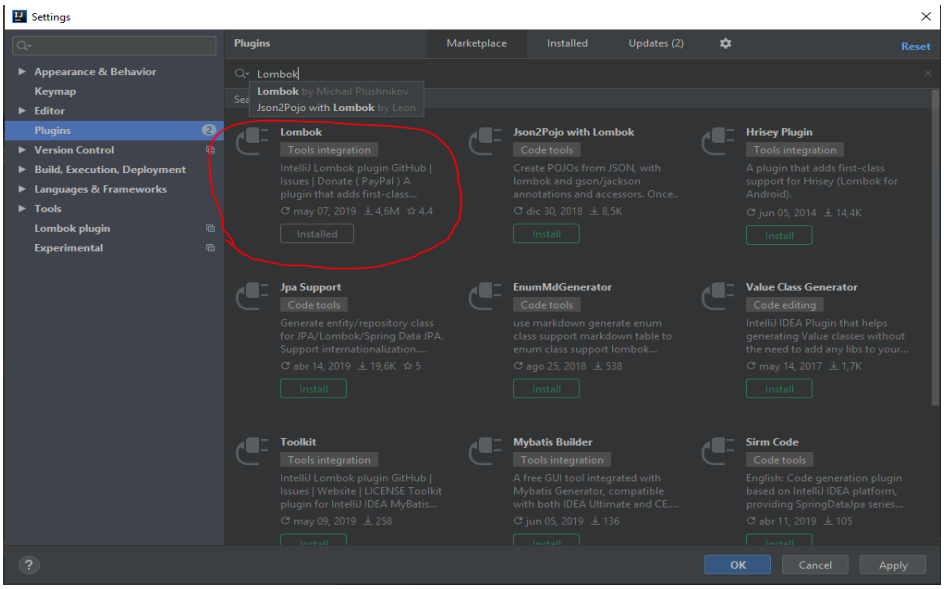
Para la creación de las clases de objetos POJO que definen nuestros objetos, se ha utilizado la herramienta LOMBOK. Lombok consiste en una librería para Java que nos ofrece, a través de anotaciones, reducir el código de las clases POJO de nuestra aplicación.

Para poder usar esta librería, ha de definirse como dependencia en el archivo POM.xml de la aplicación:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

Además, generalmente, será necesario activar el procesador de anotaciones del IDE, así como, a veces, instalar un plugin que procese estas anotaciones:





Las anotaciones le indican a Lombok qué métodos debe generar escaneando los atributos de la clase donde haya sido utilizado en tiempo de compilación. Las anotaciones más importantes son:

@Getter ➞ Genera los métodos "get" para los atributos definidos.
@Setter ➞ Genera los métodos "set"
@AllArgsConstructor ➞ Genera un constructor con todos los atributos como argumentos
@NoArgsConstructor ➞ Genera un constructor sin argumentos
@Builder ➞Permite utilizar el patrón "builder" para crear nuevos objetos, como por ejemplo:

```
ControlVO.builder().alum_reference(controlDTO.getAlum_reference())
      .controlDate(controlDTO.getControlDate())
      .difficultComment(controlDTO.getDifficultComment())
      .difficultValuation(controlDTO.getDifficultValuation())
      .id(controlDTO.getId())
      .task_reference(controlDTO.getTask_reference())
      .workLoadComment(controlDTO.getWorkLoadComment())
      .workLoadValuation(controlDTO.getWorkLoadValuation())
      .build();
```

Generalmente, los datos se intercambian con formato "JSON" o "XML" entre otros. Luego, estos datos son manejados por el controlador de la API y son traducidos al contexto del lenguaje orientado a objetos.

```
{
  "alum_reference": "string",
  "controlDate": "2019-06-10T18:19:51.802Z",
  "difficultComment": "string",
  "difficultValuation": 0,
  "id": 0,
```

```
  "task_reference": "string",
  "workDoneQuantityComment": "string",
  "workDoneQuantityValuation": 0,
  "workLoadComment": "string",
  "workLoadValuation": 0
}
```

Más tarde, interiormente, los servicios encargados manejar las peticiones llegadas al controlador del API, transforman y manejan esos objetos según las necesidades de la aplicación.

Por ello, la traducción de los datos de DTO a VO se ha de realizar en la capa de Servicio de nuestra aplicación. Esta recibirá los DTOs del controlador y guardará los objetos VO a través de la capa de datos o repositorio.

En nuestra aplicación, se ha seguido este patrón para la conversión y mapeo de datos entre objetos DTO y objetos VO o entidad. Uno de estos convertidores podemos verlo en el siguiente ejemplo:

```
@Component
public class ControlConverter {

    public ControlDTO convertEntityToDTO(ControlVO controlVO) {
        return
ControlDTO.builder().alum_reference(controlVO.getAlum_reference())

.controlDate(controlVO.getControlDate()).difficultComment(controlVO.
getDifficultComment())

.difficultValuation(controlVO.getDifficultValuation()).id(controlVO.
getId())

.task_reference(controlVO.getTask_reference()).workLoadComment(contr
olVO.getWorkLoadComment())

.workLoadValuation(controlVO.getWorkLoadValuation()).build();
    }

    public ControlVO convertDTOToEntity(ControlDTO controlDTO) {
        return
ControlVO.builder().alum_reference(controlDTO.getAlum_reference())

.controlDate(controlDTO.getControlDate()).difficultComment(controlDT
O.getDifficultComment())

.difficultValuation(controlDTO.getDifficultValuation()).id(controlDT
O.getId())

.task_reference(controlDTO.getTask_reference()).workLoadComment(cont
rolDTO.getWorkLoadComment())

.workLoadValuation(controlDTO.getWorkLoadValuation()).build();
    }
```

```
}
```

Spring Data: MongoDB

Como ya se ha comentado anteriormente, Spring Framework proporciona una potente interfaz para el manejo de datos con las principales tecnologías de base de datos. Estas interfaces se aúnan bajo el módulo de Spring-Data, el cual se subdivide en distintas librerías de las que el desarrollador dispone.

Para MongoDB, Spring Data tiene dos formas de abordar esta estructura de datos. Una es utilizando su implementación a través de la interfaz MongoRepository. Y la otra es utilizando el java-bean MongoTemplate, el cual ofrece una implementación totalmente configurable por el desarrollador a través de código o XML.

Para el uso de MongoRepository tan solo es necesario crear una interfaz que extienda de esta clase:

```
@Repository
public interface TaskRepository extends MongoRepository<TaskVO,
String>, CustomTaskRepository {
}
```

Spring Data MongoRepository nos ofrece la posibilidad de definir métodos en la interfaz que hemos creado que extiende al repository, que Spring mapeará como queries de mongo.

Por otro lado, MongoTemplate nos permite realizar un abordamiento más clásico para la creación de un repositorio. En este caso, tendremos la posibilidad utilizar un java-bean llamado MongoTemplate el cual nos es ofrecido por Spring Data y el cual es cargado en el contexto de Spring.

Esta interfaz nos permite implementar los métodos que nos sean necesarios, creando queries propias y funciones complejas para el manejo de datos con mongo.

```
public interface CustomTaskRepository {

    List<TaskVO> findAllFilteredByQuery(final Query query);

}
```

```
public class CustomTaskRepositoryImpl implements
CustomTaskRepository {
    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public List<TaskVO> findAllFilteredByQuery(final Query query) {
        return mongoTemplate.find(query, TaskVO.class);
    }
```

```
}
```

En el caso de necesitar ambas implementaciones, es necesario crear una interfaz que extienda de MongoRepository y al mismo tiempo realizar una implementación propia de algunos métodos utilizando el java-bean MongoTemplate, como se puede ver en el ejemplo anterior.

Spring Data: PostgreSQL

Se crea una clase Interfaz que extienda a "JPARepository<R,T>" donde "R" es el tipo Java de la ID y "T" el tipo de los objetos que se van a guardar. La implementación de JPA hace uso de algunos ORMs como Hibernate.

`@Entity` → Señala que esta clase es un objeto que se puede persistir en la estructura de datos.

`@Id` → Señaliza que será la ID del objeto a persistir

`@GeneratedValue` → Genera automáticamente la id cuando un objeto se persiste

Para las relaciones Uno a Muchos unidireccionales, la estructura básica de anotación es:

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
```

```
@JoinColumn(name = "employee")
```

```
private List<WorkExperienceVO> workExperiences;
```

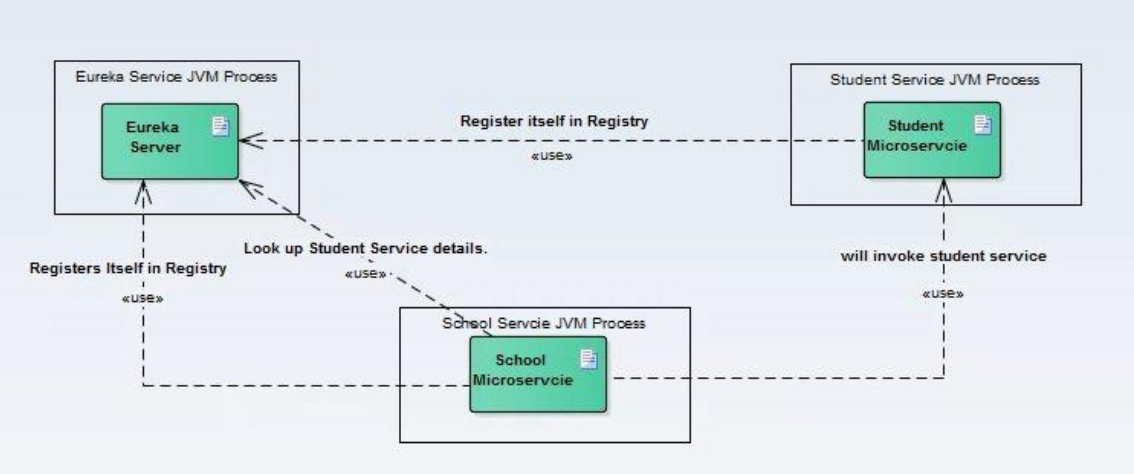
Donde "**@OneToMany"** señala qué tipo de relación es, el "cascadeType" hace referencia a la forma de comportarse la "cascada" de acciones de los datos. Es decir, si se borra un objeto de la lista de objetos referenciados también se borrará su persistencia y etc. En este caso, "ALL" hace referencia a que están todos los modos activados.

El atributo "orphanRemoval" indica si se desea que cuando se borre un objeto padre persistido, también se borren todos sus hijos.

Service Discovery

El Service Discovery es un proyecto en sí mismo basado en Spring Boot y Spring Cloud. Lo que hace es resolver las peticiones a los servicios de una aplicación compuesta por microservicios. Esto quiere decir, que es el encargado de llamar a las direcciones (endpoints o uris) de cada servicio.

Dicho de otra forma, será el que tenga la "libreta de direcciones" de los demás servicios que componen la aplicación. Es el encargado de buscar a qué dirección corresponde el servicio que se está tratando de consumir y ofrecer dicho servicio a la petición tras ser recibida.



En Spring, se utilizan las librerías de Spring Cloud, concretamente, éstas se llaman Eureka. Estas librerías te ofrecen las funcionalidades para la implementación de un Service Discovery bastante completo para una aplicación basada en arquitectura de microservicios e implementada con Spring.

Para ello, ha de crearse un proyecto Spring que tenga las dependencias de Spring Cloud dentro. Además, en el main de la aplicación Spring Boot, ha de aparecer la siguiente anotación:
`@EnableEurekaServer`

En cuanto al archivo properties de la aplicación, ha de tener una forma similar a: `server.port=8761`

`eureka.client.register-with-eureka=false`

`eureka.client.fetch-registry=false`

Esto nos permitirá acceder a una consola de información sobre el Service Discovery, con información sobre los servicios a los que hace referencia y a la máquina en la que está corriendo.

De otra forma, también puede utilizarse la configuración en formato ".yml", la cual es más limpia y más legible. Entre estas propiedades que se definen, se pueden configurar elementos del tipo:

```
server:
```

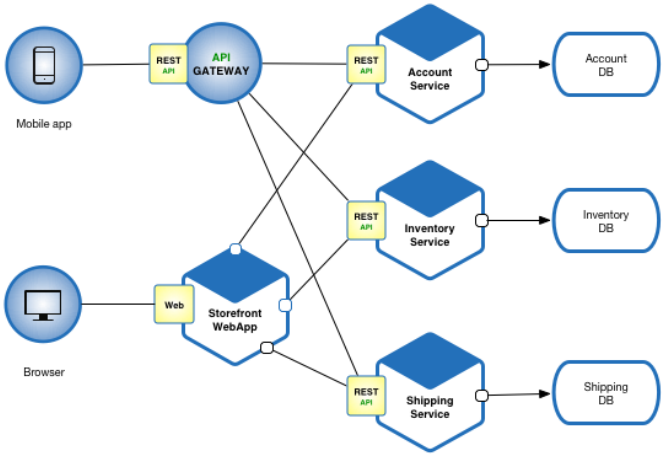
```
port: ${PORT:8761}
```

Donde la notación con el dólar, indica el uso de una variable global ya definida. En este caso, con la notación "*DEFAULT*:*new*", se está indicando que primeramente se buscará la variable definida por defecto "PORT" para la propiedad server. En caso de no existir, los dos puntos indicar un operador lógico "OR" que indica qué puerto ha de usarse en caso de no existir uno por defecto ya definido. Esto nos sirve para parametrizar la configuración de nuestra aplicación.

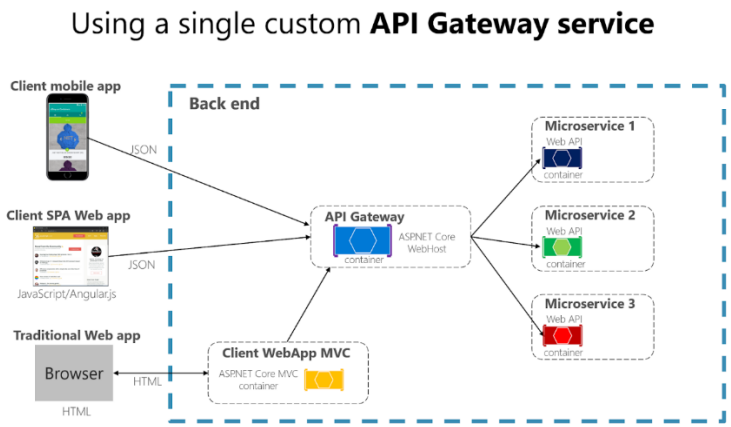
Spring Cloud Gateway:

El Gateway es un servicio basado en Spring Boot y Spring Cloud. Resumidamente, se encarga de resolver las peticiones realizadas a la aplicación de forma centralizada y distribuirla entra los servicios de una aplicación compuesta por microservicios.

Esto quiere decir, que es el encargado de recibir y organizar las peticiones hacia los distintos recursos entre los distintos servicios.



El Gateway, como su nombre indica, hace de “portero” en la conexión entre las peticiones externas y los microservicios que componen nuestra aplicación.



Para la implementación del Gateway, como para otros servicios ya creados, primero se ha generado un proyecto inicializado desde la web spring-initializr. En este caso, es necesario añadir las dependencias referentes a la librería spring cloud y el artefacto gateway:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

A estas dependencias, debemos sumarles las nombradas anteriormente para que nuestro servicio solicite su configuración al servidor config de forma ordenada y, además, que se registre en nuestro service discovery.

En esta ocasión, a configuración del servicio deberá ser algo más exhaustiva, ya que es necesario que le especifiquemos los servicios a los que debe redirigir de forma automática las peticiones que se le realicen a través de su URI.

Para nuestro caso, se han implementado dos configuraciones correspondientes a los perfiles Spring declarados, en nuestro caso, local y develop.

Para el caso del perfil local:

bootstrap-local.yml →

```
spring:
  application:
    name: gateway
  cloud:
    config:
      enabled: false
eureka:
  client:
    enabled: false
```

application-local.yml →

```
spring:
  cloud:
    gateway:
      routes:
        - id: task-calendar-service #Order service route declaration
          uri: "http://localhost:8095"
          predicates:
            - Path=/calendar/** #Path to access the service
          filters:
            - StripPrefix=1

        - id: task-control-service #Task-Control service route
declaration
```

```
          uri: "http://localhost:8098"
          predicates:
            - Path=/task-control/** #Path to access the service
          filters:
            - StripPrefix=1
```

En este caso, la configuración inicial, o "bootstrap", nos indica que no se desea que el servicio busque su configuración en el servidor y que, además, no se registre en el servidor Eureka. Esto se debe a que, en un entorno local, quizás no es deseable levantar todos los servicios, y solo es necesario ejecutar los servicios individualmente para probar su funcionamiento.

Junto a esta configuración inicial, se encuentra la configuración principal de la aplicación, en la que, en este caso, se declaran las rutas que nuestro servicio ha de redirigir junto con los predicados a los que el usuario tendrá acceso.

Spring Cloud Config Server

Spring, ofrece una implementación del llamado Config Server muy útil y sencilla. En nuestro caso, se ha creado un proyecto con la herramienta antes mencionada "spring-initializr", con las dependencias correspondientes a Spring Cloud, que es la librería padre que nos ofrece Spring Cloud Config Server:

```
<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>

</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Una vez hecho esto, tendremos un proyecto sencillo con una clase main que se ejecutará el contexto de spring. A esta clase, habrá que añadirle obligatoriamente la siguiente anotación:
`@EnableConfigServer`

Además, para este servicio es necesario que implementemos una configuración concreta en el archivo "application.yml", que debe ser la siguiente:

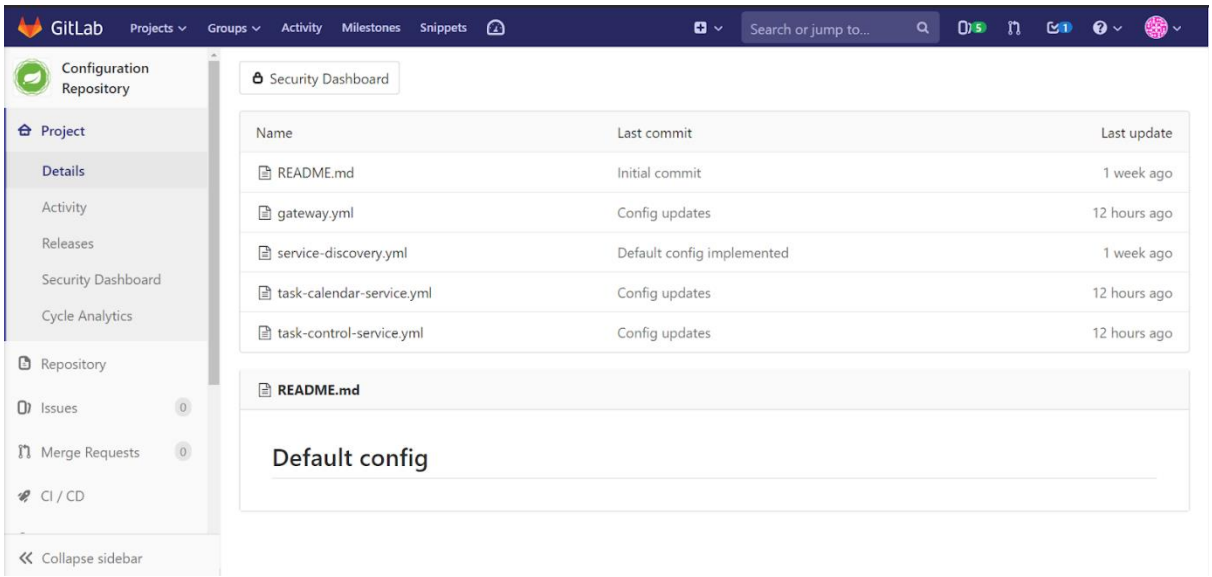
```
server:
  port: 8888

spring:
  application:
    name: server-config
  cloud:
    config:
      server:
        git:
          uri: https://gitlab.com/task-control-app-config/config-
repo.git
          force-pull: true
          clone-on-start: true
          default-label: master

          repos:

            develop:
              uri: https://gitlab.com/task-control-app-
config/config-repo.git
              force-pull: true
              clone-on-start: true
              default-label: develop

            local:
              uri: https://gitlab.com/task-control-app-
config/config-repo.git
              force-pull: true
              clone-on-start: true
              default-label: local
```

Donde podremos definir distintos repositorios asociados a los perfiles de la aplicación spring. Por ejemplo, en nuestro caso, se han definido dos repositorios, que corresponden a dos perfiles definidos: local y develop.

En el caso del repositorio local, la configuración se establece para un entorno de pruebas local en el que se pretende desplegar todos los servicios de forma local. Por otro lado, en el perfil develop, tenemos un repositorio en el que se encontrará la configuración para el despliegue en un entorno más avanzado posiblemente en remoto.

Para que cada servicio pueda encontrar su archivo de configuración, estos archivos deben de tener como nombre, el nombre del servicio y extensión ".yml" o ".properties".

En nuestro caso, se ha creado un repositorio GIT donde se contienen todos estos archivos:



Para que el resto de los servicios puedan encontrar su configuración apuntando a este servicio, también han de tener una configuración previa a su ejecución, en la que se definirán ciertos aspectos como el puerto o la dirección donde se encuentra el servicio, que no se encuentra registrado en el Service Discovery ya que es un servicio extra y totalmente independiente, y el nombre con el que buscará su configuración en el repositorio.

Por ejemplo, en el caso del servicio Gateway, tendremos una configuración en el archivo "bootstrap.yml" de la siguiente forma:

```yaml
spring:
  application:
    name: gateway
  cloud:
    config:
      name: gateway
      retry:
        max-attempts: 10
        initial-interval: 5000
      fail-fast: true
```

Con esta configuración, estamos definiendo el nombre del servicio, junto con una configuración mínima para la política de reintentos en caso de no encontrar el servicio en la dirección determinada al inicio de la aplicación. Por defecto el puerto en el que el servicio buscará el servidor de configuración es el "8888".

Además, será necesario que el servicio que desee obtener su configuración del servidor tenga varias dependencias necesarias, como son:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

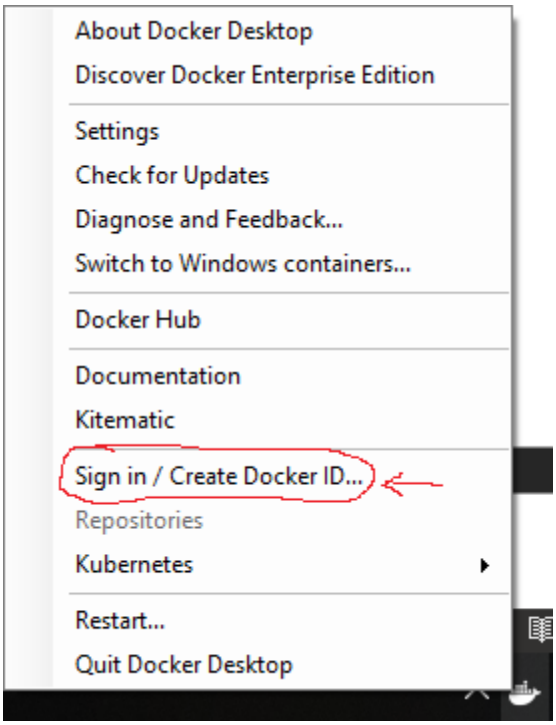
Despliegue

Docker: Creación y ejecución de imágenes

Para la creación de las imágenes Docker de los servicios que se han desarrollado, se ha investigado la herramienta desarrollada para maven llamada **fabric8 plugin**. Esta herramienta se encarga de la generación automática de la imagen Docker configurada a través de los archivos POM de cada servicio.

Previamente es necesario instalar la aplicación **Docker desktop**, o lo que es lo mismo, la aplicación de escritorio de Docker. Se puede accedes a su descarga a través del siguiente enlace ➞ https://www.docker.com/get-started, y aunque su uso es gratuito, será necesario que nos registremos para poder descargarlo.

Una vez instalado Docker, solo será necesario instalarlo para poder empezar a utilizarlo y, además, deberemos logearnos para poder descargar imágenes Docker y utilizarlas en tu propósito.



Se ha de colocar una configuración básica en el POM del proyecto para definir el plugin de fabric8 en maven.

```
<build>
    <plugins>
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>fabric8-maven-plugin</artifactId>
            <version>3.5.41</version>
        </plugin>
    </plugins>
</build>
```

Una vez hecho esto, con una configuración mínima sería suficiente, y podremos hacer la generación de la imagen Docker de nuestro proyecto.

Primero, es necesario ejecutar el comando "**mvn clean install"** para instalar la librería, o nuestro proyecto compilado, en el repositorio maven local.

Por otro lado, para generar la imagen docker se utiliza en la línea de comandos de "**mvn fabric8:build"**. Para esto no es necesaria ninguna configuración previa, pero, el plugin tiene internamente dos "modos" de generación para la imagen: uno pensado en Kubernetes y otro en OpenShift.

En el caso de Kubernetes, se trata de un modo “simple” en el que se crea una imagen Docker típica. Aunque se puede configurar el modo de generación manualmente, para crear una imagen normal de Docker no es necesario.

En el caso por defecto, el plugin consulta su API automáticamente la cual indica en cada caso qué configuración es la recomendada. Casi siempre, por defecto, es el modo “Kubernetes” el cual genera una imagen Docket genérica.

Generalmente, para la creación de las imágenes, la configuración previa se define en el POM cuando se define el plugin de fabric. Aunque también existe la opción de incluir, manualmente los archivos o archivo DockerFile, para lo cual se le deben indicar la/las rutas.

Comandos Importantes:

docker images → Muestra todas las imágenes conocidas

docker ps → Muestra todos los contenedores en ejecución

docker run -p PORT:PORT image_name → Ejecuta una imagen Docker dentro de un contenedor creado según las especificaciones de la imagen en sí misma.

docker stop <CONTAINER_ID> → Para la ejecución del contenedor con la ID introducida

Existen otras formas de generar una imagen Docker, como por ejemplo, definiendo un DockerFile y generando la imagen a través de Docker, o lo que es lo mismo, manualmente.

Existen otras formas y herramientas, como es el caso de la herramienta o plugin desarrollada por Spotify la cual permite, de forma sencilla, generar una imagen Docker con una configuración sencilla.

Aun así, en este caso, se utiliza el plugin fabric8 ya que además de generar la imagen Docker, permite desplegar o subir la misma directamente a Kubernetes u a Openshift además de estar orientado a microservicios y otras tecnologías muy actualizadas. Esto que implica que su configuración avanzada sea algo más tediosa, aunque con un mayor abanico de posibilidades.

Compilación y ejecución del proyecto a través de Maven

En resumen, para la compilación y ejecución de los servicios de forma individual a través de Maven, necesitaremos ejecutar los siguientes pasos:

Primero, ejecutaremos el comando **mvn clean install**

Luego, una vez se haya realizado la compilación de forma correcta y sin errores, podremos ejecutar la aplicación. En este caso tendremos dos opciones:

mvn springboot:run

mvn springboot:run -Dspring.profiles.active=local

Despliegue a través de Docker

Una vez tengamos las imágenes necesarias para nuestro proyecto, tendremos dos opciones:

La primera consiste en levantar cada servicio de forma individual con los comandos antes mencionados. En este caso, primero deberemos conocer el nombre de nuestra imagen con el comando **docker images** para luego ejecutar el comando **docker run** con las variables necesarias como por ejemplo, los puertos de su ejecución: **docker run -p 8080:8080 gateway**.

La segunda opción, es algo más avanzada, y requiere del uso de una herramienta más compleja llamada Docker-compose. En nuestro caso, esta será la mejor opción, y el archivo Docker-compose de nuestra aplicación tendrá el siguiente aspecto:

```
version: '3'
services:

  postgresdb:
    image: postgres:latest
    container_name: postgresdb
    volumes:
      - /private/var/lib/postgresql:/var/lib/postgresql
    environment:
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_USER=postgres
      - POSTGRES_DB=postgres
    ports:
      - "5432:5432"
    expose:
      - 5432

  mongodb:
    image: mongo:latest
    container_name: mongodb

  server-config:
    image: mtribor/taskcontrolapp-server-config:latest
    container_name: server-config
    ports:
      - "8888:8888"
    expose:
      - 8888


  service-discovery:
    image: mtribor/taskcontrolapp-service-discovery:latest
    container_name: service-discovery
    ports:
      - "8761:8761"
    expose:
      - 8761
    depends_on:
```

```
      - server-config
    links:
      - server-config
    environment:
      SPRING_CLOUD_CONFIG_URI: http://server-config:8888
      EUREKA_URI: http://service-discovery:8761/eureka



  task-control-service:
    image: mtribor/task-control-service:latest
    container_name: task-control-service-container
    expose:
      - 8098
    ports:
      - "8098:8098"
    depends_on:
      - postgresdb
      - server-config
      - service-discovery
    links:
      - server-config
      - service-discovery
    environment:
      SPRING_CLOUD_CONFIG_URI: http://server-config:8888
      EUREKA_URI: http://service-discovery:8761/eureka



  task-calendar-service:
    image: mtribor/task-calendar-service:latest
    container_name: task-calendar-service-container
    ports:
      - "8095:8095"
    expose:
      - 8095
    depends_on:
      - mongodb
      - server-config
      - service-discovery
    links:
      - mongodb
      - server-config
      - service-discovery
    environment:
      SPRING_DATA_MONGODB_URI: mongodb://mongodb/task-calendar-
service_db
      SPRING_CLOUD_CONFIG_URI: http://server-config:8888
      EUREKA_URI: http://service-discovery:8761/eureka



  gateway:
```

```
    image: mtribor/taskcontrolapp-gateway:latest
    container_name: gateway
    ports:
      - "8080:8080"
    depends_on:
      - server-config
      - service-discovery
    links:
      - server-config
      - service-discovery
    environment:
      SPRING_CLOUD_CONFIG_URI: http://server-config:8888
      EUREKA_URI: http://service-discovery:8761/eureka
```

Para la ejecución de la aplicación a través de este método, será necesario navegar, a través de la consola de comandos, hasta la carpeta o directorio donde se encuentre dicho archivo y, una vez allí, deberemos ejecutar el comando **docker-compose up.**

Conclusión

Desde el comienzo de esta investigación, como alumno, he sabido que se trataba de un proyecto muy ambicioso ya que, más que centrado en el desarrollo, se trata de un tema de investigación centrado en la capa más alta del diseño software: **la arquitectura.**

Esto, sin duda me ha ayudado a crecer como desarrollador, ya que ha sido un salto inmenso respecto a las enseñanzas aportadas en el centro educativo debido a la necesidad de entender una serie de tecnologías y conceptos tan actuales, que en algunos casos sus definiciones son etéreas y poco accesibles.

Como conclusión personal, quizás esto sea el mejor resumen: <<*Parecía demasiado para mí pero no ha sido para tanto. Dividir y abstraer no son conceptos nuevos, y los microservicios lo saben y lo han utilizado a su favor y con ellos he crecido yo como desarrollador*>>.

Respecto a las conclusiones de la investigación, centradas en la tecnología, posiblemente estas sean las mayores ventajas por las que me atrevería a decir que este patrón de arquitectura es recomendable:

- Integrar y escalar con aplicaciones de terceros es muy sencillo
- Un proyecto modular basado en microservicios evoluciona de forma más natural, es una forma fácil de gestionar diferentes desarrollos, utilizando los recursos disponibles, al mismo tiempo.
- Los microservicios pueden desplegarse según sea necesario, por lo que funcionan bien dentro de metodologías ágiles.

- El mantenimiento es más simple y barato — con los microservicios se puede hacer mejoras de un módulo a la vez, dejando que el resto funcione normalmente.
- Las soluciones desarrolladas con arquitectura de microservicio permiten la mejora rápida y continua de cada funcionalidad.
- Versátil — los microservicios permiten el uso de diferentes tecnologías y lenguajes.
- El desarrollador puede aprovechar las funcionalidades que ya han sido desarrolladas por terceros — no necesita reinventar la rueda, simplemente utilizar lo que ya existe y funciona.

Con estas premisas, los microservicios son una arquitectura que sin duda tienen mucho sentido hoy día, donde la tecnología está cada vez más descentralizada y la mayor parte de las aplicaciones requieren del manejo de grandes datos y soportar grandes cargas de trabajo.

Y por suerte, para eso está la nube. La nube nos ofrece esa escalabilidad que las aplicaciones necesitan, y esa conectividad con el mundo que los usuarios necesitamos, pero no todo es tan bonito.

Las aplicaciones monolíticas no son capaces de sacarle todo el partido a las tecnologías cloud, ya que usualmente estamos acostumbrados a trabajar con servidores estáticos, con direcciones fijas donde se alojan nuestras aplicaciones.

Pero para eso están los microservicios, para definir una nueva arquitectura de comunicación que nos permita unificar todos esos servicios deslocalizados a través de unos protocolos de red que transmitan los datos y, mientras, la nube que haga el resto con eso de equilibrar las cargas y flexibilidad las capacidades de nuestros servidores.

Por otro lado, también existen contras, y son los siguientes:

- Debido a que los componentes están distribuidos, las pruebas globales son más complicadas.
- Es necesario controlar el número de microservicios que se gestionan, ya que cuantos más microservicios existan en una solución, más difícil será gestionarlos e integrarlos.
- Los microservicios requieren desarrolladores experimentados con un nivel muy alto de experiencia.
- Se requiere un control exhaustivo de la versión.
- La arquitectura de microservicios puede ser costosa de implementar debido a los costos de licenciamiento de aplicaciones de terceros.

Repositorio GitHub con todo el contenido

https://github.com/MTribor/DAM_PFG_Microservicios

Bibliografía

Microservicios

- Tutorial sobre Microservicios con ejemplo → https://www.guru99.com/microservices-tutorial.html
- Ejemplo de aplicación de una tienda hecha con arquitectura en microservicios y distintas tecnologías para cada microservicio - Instana Robot Shop → https://github.com/instana/robot-shop/ || Guía para esta aplicación → https://dzone.com/articles/stans-robot-shop-a-sample-microservice-application
- Ejemplo de aplicación de tienda online basada en microservicios - Maltorpro Microservice Shop Demo → https://github.com/maltorpro/microservice-shop-demo
- Ejemplo sencillo de tienda online con dos microservicios - IBM → https://www.ibm.com/blogs/bluemix/2015/03/sample-application-using-microservices-bluemix/?s_tact=C43401LW
- Ejemplo de tienda online con API documentada y todo tipo de documentos de ayuda - Weaveworks Sock Shop → https://microservices-demo.github.io/
- Artículo sobre la arquitectura de Microservicios - → https://microservices.io/patterns/microservices.html
- Guía sencilla para crear microservicios - DZone→ https://dzone.com/articles/quick-guide-to-microservices-with-spring-boot-20-ehttps://dzone.com/articles/quick-guide-to-microservices-with-spring-boot-20-e
- ¿Por qué usar arquitectura de Microservicios? Utilizando un API Gateway- Microsoft → https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern

Spring Cloud

Gateway

- Mini-guía de Spring Cloud Gateway - Bi Geek → https://blog.bi-geek.com/arquitecturas-microservicios-spring-cloud-gateway/
- ¿Por qué usar arquitectura de Microservicios? Utilizando un API Gateway- Microsoft → https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern

Service Discovery

- Documentación Spring Service Discovery Netflix - web oficial → https://cloud.spring.io/spring-cloud-netflix/multi/multi__service_discovery_eureka_clients.html

Config Server

- Spring cloud config client retry → https://cloud.spring.io/spring-cloud-config/1.4.x/multi/multi__spring_cloud_config_client.html
- Spring cloud retry guide → https://www.alessiofiore.com/blog/programming/j2ee/spring/1654/how-to-retry-on-spring-config-server-connection-failure/

Spring Boot

- Variables de entorno Spring - StackOverflow → https://stackoverflow.com/questions/47580247/optional-environment-variables-in-spring-app
- Construir una API REST con Spring y Java → https://www.baeldung.com/building-a-restful-web-service-with-spring-and-java-based-configuration
- Tutoriales RESTFul con Spring → https://www.baeldung.com/rest-with-spring-series

Spring Data

MongoDB

- Implementación de un repositorio personalizado → https://stackoverflow.com/questions/50185775/can-we-use-both-mongorepository-and-mongotemplate-in-same-application || https://blog.marcnuri.com/spring-data-mongodb-implementacion-de-un-repositorio-a-medida/ ||

https://stackoverflow.com/questions/17008947/whats-the-difference-between-spring-datas-mongotemplate-and-mongorepository

PostgreSQL

- Spring data y PostgreSQL → https://dzone.com/articles/spring-boot-and-postgresql
- Persistencia con Spring Data y PostgreSQL → http://codedpoetry.com/persistence-with-spring-data-postgresql/

Docker

- Documentación Fabric8 (maven plugin): https://maven.fabric8.io/ - https://dmp.fabric8.io/#simple-dockerfile-build - https://fabric8.io/guide/mavenDockerBuild.html - https://github.com/fabric8io/docker-maven-plugin/blob/master/doc/intro.md
- Crear un DockerFile → https://docs.docker.com/get-started/part2/
- Crear un DockerFile para Spring Boot → https://spring.io/guides/gs/spring-boot-docker/ - https://spring.io/guides/topicals/spring-boot-docker
- Comandos para Docker → https://medium.com/the-code-review/top-10-docker-commands-you-cant-live-without-54fb6377f481 - https://coderwall.com/p/ewk0mq/stop-remove-all-docker-containers - https://linuxize.com/post/how-to-remove-docker-images-containers-volumes-and-networks/
- Durante la investigación de este proyecto se redactó un artículo referente a fabric8 por los alumnos Marco Borreguero y Alberto Serrano ➞ https://profile.es/blog/fabric8-gestion-imagenes-y-contenedores-docker-desde-maven/