Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de material didáctico para el aprendizaje de Spring

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alejandro Lahiguera Hervás

Tutor: José Vicente Busquets Mataix

Curso 2020-2021

Resumen

Actualmente Spring es uno de los frameworks más usados en el mundo y tiene una gran demanda en el mercado laboral. La manera en la que simplifica la creación de aplicaciones ha favorecido su popularidad y crecimiento. Por ello, se va a desarrollar una guía didáctica para facilitar el aprendizaje de este framework. La intención de este proyecto es describir y mostrar con detalle los puntos más importantes, eliminar la información innecesaria y seguir un camino correcto hacia la creación y el entendimiento de aplicaciones basadas en este framework. Toda la información estará enfocada hacia un uso en el mercado laboral con ejemplos prácticos reales.

Palabras clave: Spring, Spring Boot, backend, framework, MVC, guía, aprendizaje, DB2, JPA, gateway, eureka, thymeleaf.

Abstract

Spring is currently one of the most widely used frameworks in the world and is in great demand in the job market. The way in which it simplifies the creation of applications has favored its popularity and growth. Therefore, a didactic guide will be developed to facilitate the learning of this framework. The intention of this project is to describe and show in detail the most important points, eliminate unnecessary information and follow a correct path towards the creation and understanding of applications based on this framework. All the information will be focused on a use in the job market with real practical examples.

Keywords: Spring, Spring Boot, backend, framework, MVC, guide, learning, DB2, JPA, gateway, eureka, thymeleaf.

Resum

Actualment Spring és un dels frameworks més usats en el món i té una gran demanda en el mercat laboral. La manera en la qual simplifica la creació d'aplicacions ha afavorit la seua popularitat i creixement. Per això, es desenvoluparà una guia didàctica per a facilitar l'aprenentatge d'aquest framework. La intenció d'aquest projecte és descriure i mostrar amb detall els punts més importants, eliminar la informació innecessària i seguir un camí correcte cap a la creació i l'enteniment d'aplicacions basades en aquest framework. Tota la informació estarà enfocada cap a un ús en el mercat laboral amb exemples pràctics reals.

Paraules clau: Spring, Spring Boot, backend, framework, MVC, guia, aprenentatge, DB2, JPA, gateway, eureka, thymeleaf.

Tabla de contenidos

1. Introducción 11
1.1 Motivación 11
1.2 Objetivos 12
1.3 Estructura de la memoria 13
2. Estado del arte 14
2.1 Tecnologías transversales 14
2.2 Tecnologías backend 16
3. Retos principales 19
3.1 Servicio REST con lógica PL/SQL 19
3.2 Thymeleaf como motor de plantillas HTML 19
3.3 Unificar el enrutamiento con un gateway 20
3.4 Registrar y localizar nuestros microservicios 20
3.5 Evitar validaciones de otras entidades 20
3.6 Automatizar la creación de la estructura de test 21
4. Diseño de la arquitectura 22
4.1 Base de datos 22
4.2 Patrón MVC e inyección de dependencias 24
4.3 API Gateway 27
5. Desarrollo 29
5.1 API Rest con Spring 29
5.2 Funciones y procedimientos almacenados 41
5.3 Implementación de Thymeleaf 44
5.4 API Gateway con Spring 46
5.5 Eureka Server con Spring 48
6. Implementación 51
6.1 Instalación de DB2 51
6.2 Creación de microservicios 52
6.3 Dependencias 54
6.4 Configuración inicial 56
7. Pruebas 58
7.1 Esquema de las validaciones 58

7.2 Pruebas unitarias 60
7.3 Desactivar validaciones externas 63
8. Conclusiones y trabajos futuros 65
8.1 Conclusiones.. 65
8.2 Trabajos futuros 66
9. Referencias 69

Índice de figuras

Figura 1. *Definición de Spring Boot* 14
Figura 2. *Ejemplo de interfaz de Fork* 16
Figura 3. *Ejemplo de interfaz de SqlDbx* 17
Figura 4. *Ejemplo de una función en DB2* 22
Figura 5. *Ejemplo de procedimiento almacenado en DB2* 23
Figura 6. *Esquema patrón MVC* 24
Figura 7. *Esquema de la inyección de dependencias en Spring* 26
Figura 8. *Arquitectura microservicios sin API Gateway* 27
Figura 9. *Arquitectura microservicios con API Gateway* 28
Figura 10. *Uso del ORM en Spring* 29
Figura 11. *Implementación de una super clase* 30
Figura 12. *Clave primaria compuesta* 31
Figura 13. *Inyección de la clave primaria compuesta* 31
Figura 14. *Entidad extendida de ejemplo* 32
Figura 15. *Relación @OneToOne* 32
Figura 16. *Relación @OneToMany* 32
Figura 17. *Relación @ManyToOne para satisfacer @OneToMany* 32
Figura 18. *Múltiples columnas en una unión* 33
Figura 19. *Relación @ManyToMany en usuarios* 33
Figura 20. *Relación @ManyToMany en roles* 33
Figura 21. *Modelo de ejemplo* 34
Figura 22. *Converter de ejemplo* 35
Figura 23. *Repositorio de ejemplo* 36
Figura 24. *Consulta SQL desde Java para acceder a los datos* 36
Figura 25. *JPQL snippet* 37
Figura 26. *Conversor automático entidad-modelo* 37
Figura 27. *Ejemplo de inyección de dependencias de un repositorio en un servicio* 38
Figura 28. *Servicio de ejemplo* 38
Figura 29. *Controlador de ejemplo* 39
Figura 30. *Métodos GET y POST de ejemplo de un controller* 39
Figura 31. *Petición GET* 40
Figura 32. *Resultado de la petición GET* 40
Figura 33. *Petición POST* 40
Figura 34. *Entidad falsa para llamar a una función* 41
Figura 35. *Repositorio con consulta SQL para llamar a una función DB2 desde Java* 41
Figura 36. *Inyección de dependencias de la función-repositorio* 42
Figura 37. *Ejecución de la función desde Java* 42
Figura 38. *Anotaciones para la entidad del procedimiento almacenado* 42
Figura 39. *Repositorio que implementa un procedimiento almacenado en DB2* 43
Figura 40. *Ejemplo de controlador Thymeleaf 1* 44
Figura 41. *Ejemplo de controlador Thymeleaf 2* 44
Figura 42. *Código HTML usando atributos Thymeleaf* 45
Figura 43. *Formulario creado con Thymeleaf* 45
Figura 44. *Archivo de configuración application.yml del API Gateway* 47
Figura 45. *Dependencia Eureka Client* 48

Figura 46. *Configuración básica del Eureka Server* 48
Figura 47. *Dashboard Spring Eureka 1* 49
Figura 48. *Dashboard Spring Eureka 2* 49
Figura 49. *Dashboard Spring Eureka 3* 49
Figura 50. *Instalación DB2 – Descargar el instalador* 51
Figura 51. *Instancia DB2 en los iconos ocultos de la barra de tareas* 52
Figura 52. *Consulta de la base de datos SAMPLE con SqlDbx* 52
Figura 53. *Interfaz de Spring Initializr* 53
Figura 54. *Lista de dependencias mencionadas* 55
Figura 55. *Estructura base de los microservicios* 56
Figura 56. *Esquema de la estructura de las validaciones* 58
Figura 57. *Consulta que genera una nueva clave primaria* 60
Figura 58. *Métodos de las validaciones* 60
Figura 59. *Interfaz servicio validaciones para los test* 60
Figura 60. *Ejemplo de nuestra primera validación* 61
Figura 61. *Inyección de dependencia del servicio de validaciones* 61
Figura 62. *Validación inyectada en la entidad extendida* 61
Figura 63. *Ejemplo de fixture* 62
Figura 64. *Ejemplo de test que comprueba nuestra validación* 62
Figura 65. *Desactivación del resto de servicios de validaciones* 64
Figura 66. *Llamada antes de cada test a la desactivación de las validaciones* 64
Figura 67. *Ejemplo de tareas Gradle para automatizar* 66
Figura 68. *Implementación de tarea Gradle* 67
Figura 69. *Resultado de la ejecución de la tarea Gradle* 67

1. Introducción

La introducción de esta memoria está dividida en tres partes formando así el primer apartado: motivación, objetivos y estructura de la memoria. La motivación establece la razón de la elección del tema, justifica la importancia de la materia y muestra el interés personal en dicha elección. Los objetivos especifican el fin al que se pretende llegar mediante procesos que se detallarán y que, obtenidos unos resultados, demuestren si se consigue o no el planteamiento inicial. Por último, la estructura de la memoria describe de forma concisa cada capítulo de la memoria.

1.1 Motivación

Mis principales motivaciones para el desarrollo de este Trabajo de Fin de Grado se pueden resumir en los siguientes aspectos: haber obtenido experiencia en la realización de aplicaciones construidas en este framework, el deseo de compartir mis conocimientos con el resto para que puedan resultarles útiles a programadores que se inicien en este framework o que ya posean conocimientos y pretendan ampliarlos, la temática que comprende el proyecto en sí y la importancia de facilitar conocimiento útil, real y necesario en el mundo de la programación.

Spring, de la mano de Spring Boot, simplifica la creación de aplicaciones robustas, especialmente para el backend y lo convierte en un entorno perfecto para crecer y afianzar conocimientos previos de programación, en mi caso los de Java. Permite el desarrollo de aplicaciones de escritorio, por la línea de comando, web, microservicios, aplicaciones que acceden a bases de datos relacionales, no relacionales, a través de un ORM, en la nube, escalables…

Por este motivo, he decidido desarrollar una guía completa que abarque información práctica y actual en un único proyecto para facilitar el acceso y el entendimiento de este framework tan demandado en el mercado laboral.

Personalmente tengo interés en seguir creciendo en este framework debido a su potencial, tanto en lo laboral como en lo personal, para poder desarrollar aplicaciones en un futuro donde no dude en contar con esta tecnología si, tras un estudio del entorno, requiero de sus características. Además de esto, me gustaría agradecer que, en el ámbito de la programación, se comparta tanta información para que resulte más fácil el crecimiento de tecnologías como ésta y adquirir nuevas habilidades o mejorarlas y que dichas tecnologías sean de código abierto.

1.2 Objetivos

El objetivo principal de este proyecto consiste en desarrollar una guía completa con información y procedimientos prácticos, actualizados y relevantes que faciliten el aprendizaje del framework Spring. Para ello se tendrá que explicar de manera correcta qué tecnologías se van a usar, cuáles son las bases y características principales que posee Spring y proporcionar ejemplos prácticos para un mejor entendimiento. Adicionalmente existen unos objetivos generales y específicos que concretarán el conjunto de tareas que expresarán lo que se va a hacer:

Los objetivos generales serán los siguientes:

- Entender cómo funciona Spring y cuáles son sus características principales.
- Dar a conocer las tecnologías seleccionadas y mostrar la importancia que tienen individualmente y en su conjunto.
- Explicar cómo se implementa el estilo de arquitectura MVC de una aplicación Spring.
- Conocer las anotaciones más importantes de Spring y cuál es su uso.
- Gestionar la base de datos con sus respectivas tablas y relaciones.
- A nivel personal, manifestar mis conocimientos y detallarlos de forma que puedan resultarles útiles a cualquier persona que busque conocimiento sobre cómo ocupar un puesto de trabajo como desarrollador backend usando Spring.

Por otro lado, los objetivos específicos serán los siguientes:

- Crear procedimientos almacenados y funciones en DB2 usando PL/SQL, enseñar sus diferencias y la importancia que tienen para cualquier programador backend.
- Aprender la diferencia entre controladores, modelos, super clases, repositorios, servicios, entidades extendidas, *fixtures* y *converters* y sus características principales.
- Extraer registros de la base de datos gracias a Hibernate y JPA.
- Convertir una entidad en un modelo automáticamente gracias a Hibernate.
- Conocer qué son las super clases, las clases extendidas y su utilidad.
- Crear un servicio REST.
- Conocer las propiedades clásicas de todo proyecto Spring.
- Llamar a procedimientos almacenados y ejecutar funciones mediante repositorios.
- Evitar validaciones de otras entidades y desarrollar test que las respalden.
- Saber los conocimientos básicos de Thymeleaf y cómo crear plantillas que reciban y envíen datos al controlador.
- Crear una API Gateway para gestionar todas las redirecciones del resto de microservicios en un solo proyecto.
- Controlar qué microservicios están en marcha con Eureka Server.
- Automatizar la creación de la estructura de test con Gradle.

1.3 Estructura de la memoria

A continuación, se determinará brevemente el contenido de los capítulos en los que se organizará la memoria:

- Capítulo 1, **Introducción**. En el primer apartado se presenta la motivación que hay detrás de la elección del tema y los objetivos que se tratan de conseguir a lo largo de la memoria.
- Capítulo 2, **Estado del arte**. Se analizan y se destacan las características principales de las tecnologías que se recomiendan conocer para desarrollar lo implementado en este trabajo.
- Capítulo 3, **Retos principales**. Tras una investigación de cuáles son los problemas que deben ser capaces de resolver los programadores backend que desarrollen aplicaciones usando tecnologías Spring, se enuncia la situación de la que partimos y los retos que pretendemos superar.
- Capítulo 4, **Diseño de la arquitectura**. En este apartado, se muestra la arquitectura que seguirán los patrones que debemos conocer para llevar a cabo las implementaciones. Además, se enseña cómo usaremos nuestra base de datos.
- Capítulo 5, **Desarrollo**. Se detalla la solución de la mayoría de los problemas planteados en el apartado de Retos principales. Dado que esta guía pretende hacer capaz a todo programador que la lea de implementar las mismas soluciones, se incluye más código de lo habitual en este apartado.
- Capítulo 6, **Implementación**. Se describe los pasos a seguir para construir y configurar los microservicios necesarios para implementar las soluciones mencionadas.
- Capítulo 7, **Pruebas**. En este apartado, se explica cómo realizar validaciones en las entidades de la base de datos y cómo crear los test que confirmen la correcta implementación de estas validaciones.
- Capítulo 8, **Conclusiones y trabajos futuros**. En el último apartado de la memoria se resumen las ideas más importantes en base a los conocimientos explicados, se nombran los objetivos cumplidos y se plantean unas mejoras que pretenden ampliarnos los conocimientos en este framework.

Adicionalmente la memoria contiene una **bibliografía** con las referencias a los documentos que se han utilizado como fuente de información.

2. Estado del arte

El segundo apartado del proyecto proporciona un acercamiento a las tecnologías que se utilizarán en el trabajo, analizando la situación en la que se encuentra cada una de ellas y resaltando la importancia y las características principales que posee cada una de ellas.

2.1 Tecnologías transversales

Independientemente de la forma de trabajar con Spring, es conveniente conocer estas tecnologías ya que facilitan el desarrollo y, especialmente, la configuración inicial del proyecto para su arranque. Las herramientas que se han elegido son las siguientes:

- **Spring framework**

La primera versión de Spring se publicó en marzo de 2004, de la mano de Rod Johnson, y marcó un gran avance con respecto al modelo de programación tradicional con la inversión de control. Aunque la inversión de control no surgió con el lanzamiento de Spring, sí que es cierto que éste popularizó el concepto. Por este motivo, entre las características principales de Spring framework destaca la inversión de control implementada mediante la inyección de dependencias [1].

Inicialmente, la ejecución de los programas seguía un flujo simple y secuencial. El código se ejecutaba línea a línea hasta que aparecieron las bibliotecas con código reutilizable. Esta aparición añadió complejidad a dicho flujo. Principalmente debido a que el procedimiento había sido cambiado y se saltaba de componente en componente. Únicamente había un inconveniente: el acoplamiento entre componentes [2].

Es aquí donde aparece la implementación basada en inyección de dependencias. La ventaja que proporciona este patrón, que más tarde analizaremos en profundidad, es que el propio framework suministra objetos a una clase, en lugar de ser dicha clase quien cree dichos objetos [3]. Esto es posible gracias a que Spring posee un contenedor donde se gestionan las instancias de los objetos de la aplicación y su respectivo ciclo de vida.

- **Spring Boot**

Spring Boot es un asistente que acelera la creación de proyectos Spring. La clave de este asunto es anteponer una serie de convenciones que predominan sobre la configuración. Dicho en otras palabras, con Spring Boot se establecen ciertas configuraciones por defecto, a no ser que se especifique lo contrario. Puede decirse que Spring Boot es la suma de Spring framework más un servidor embebido, restándole la configuración XML que nos quitaría tiempo del proceso de desarrollar la lógica directamente. Su intención no es sustituir a Spring, debido a que realmente trabaja con Spring pero de una manera más sencilla. Por todos estos motivos, Spring Boot no es un framework como sí lo es Spring [4].

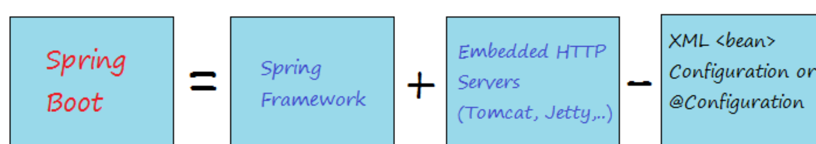

Figura 1. *Definición de Spring Boot*

- **Spring Boot Initializr**

Si a la facilidad que nos proporciona Spring Boot para agilizar la creación de aplicaciones Spring le añadimos este asistente web, nos situaremos en el mejor marco de trabajo posible. Esta aplicación web genera la estructura del proyecto Spring Boot de manera automática. Como consecuencia de su uso, que veremos en el apartado de Implementación de esta memoria, se nos facilitará notablemente la creación de microservicios [5].

- **Gradle**

Spring Boot Initializr te plantea la dicotomía de elegir entre Maven o Gradle. En nuestro caso elegiremos Gradle como bien explicaremos más adelante, aunque realmente realizan la misma función de forma distinta. El uso de esta herramienta es para automatizar la construcción del código que se desarrolla. Gradle dispone de tareas para pasar de código fuente a código ejecutable compilado entre otras. Además, es capaz de gestionar las dependencias de nuestro proyecto con pocas líneas de código.

- **Thymeleaf**

Como se ha mencionado, Spring se utilizará principalmente para construir microservicios en la parte de backend. Estos microservicios pueden implementar el patrón MVC que a lo largo de la memoria analizaremos. En caso de querer gestionar la capa de la Vista, Spring permite la integración completa del motor de plantillas Thymeleaf. Esta herramienta añade atributos al lenguaje HTML para facilitar ciertas tareas al programador. Es decir, dentro del mismo proyecto Spring podremos crear interfaces para que el usuario interaccione con nuestra aplicación de forma visual.

- **IntelliJ**

Uno de los entornos de desarrollo más populares dentro del mundo de la programación es IntelliJ. Aunque su uso más común se da para programar en Java, también soporta lenguajes específicos del frontend como podrían ser JavaScript y HTML.

Se hace saber al programador, que si es estudiante, puede solicitar una licencia de IntelliJ IDEA Ultimate que es una versión más completa de este entorno de trabajo.

- **Git**

Actualmente, es el software de control de versiones más usado en el mundo. Su desarrollo se encuentra activo y está al alcance de todos debido a que es código abierto. Principalmente permite la construcción de aplicaciones de forma conjunta entre distintos programadores. Como consecuencia de las ventajas que proporciona, dominar su uso es un pilar fundamental para poder trabajar en cualquier empresa tecnológica [6].

- **Fork**

Fork es una aplicación de escritorio que proporciona un cliente Git cuyo objetivo es gestionar el uso de nuestro controlador de versiones de forma visual. Si decidimos usar esta herramienta, podremos manejar las versiones de nuestro código mediante su interfaz [7]. Entre las distintas acciones que nos facilita se encuentran: clonar repositorios, crear ramas, eliminar ramas, descartar cambios, solucionar conflictos a la hora de unir ramas y, las más importantes, realizar *push*, *pull*, *commit* y *fetch*.

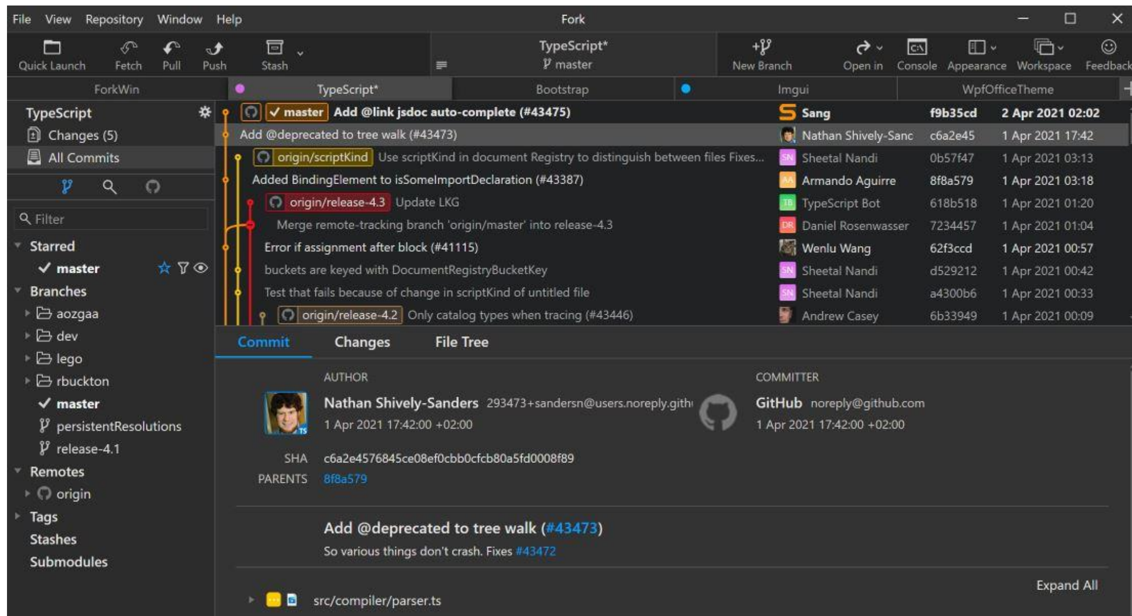

Figura 2. *Ejemplo de interfaz de Fork*

2.2 Tecnologías backend

Las tecnologías que mencionaremos a continuación son específicas para gestionar la capa de persistencia o tener acceso a ella. Aunque no son las más populares, se complementan adecuadamente para realizar todo tipo de tareas que cualquier programador de base de datos debe llevar a cabo.

- **IBM DB2**

La decisión de qué base de datos utilizar en nuestra aplicación puede resultar vital para un correcto desarrollo del software. En nuestra memoria, hemos optado por elegir IBM DB2. Aunque no se haya visto en la carrera y no sea tan popular en entornos particulares, en el mundo laboral es una de las más usadas y con gran reconocimiento por su rendimiento, fiabilidad y su capacidad de correr en diferentes plataformas operativas. Esta base de datos es del tipo relacional y utiliza SQL como lenguaje de consulta.

- **PL/SQL**

PL/SQL (Procedural Language/Structured Query Language) es un lenguaje de programación de base de datos potente, pero sencillo. Amplia SQL con elementos característicos de los lenguajes de programación: declaración de variables, bucles, control de flujo, funciones, etc. [8]. El motivo de que se encuentre en la memoria es debido a que existe la posibilidad de que en el mundo laboral se ordene programar lógica en la capa de persistencia. Por este motivo, veremos cómo crear funciones y procedimientos almacenados e integrarlos desde Spring y qué ventajas aporta.

- **SqlDbx**

Este IDE para el desarrollo y administración de bases de datos destaca por la facilidad que ofrece para trabajar en este tipo de entornos. Su interfaz es sencilla e intuitiva. Posee un editor SQL avanzado y un explorador de objetos de la base de datos. No requiere instalación, es gratuito y permite la construcción y ejecución de consultas, funciones, procedimientos almacenados, triggers, vistas, tablas, etc. [9].

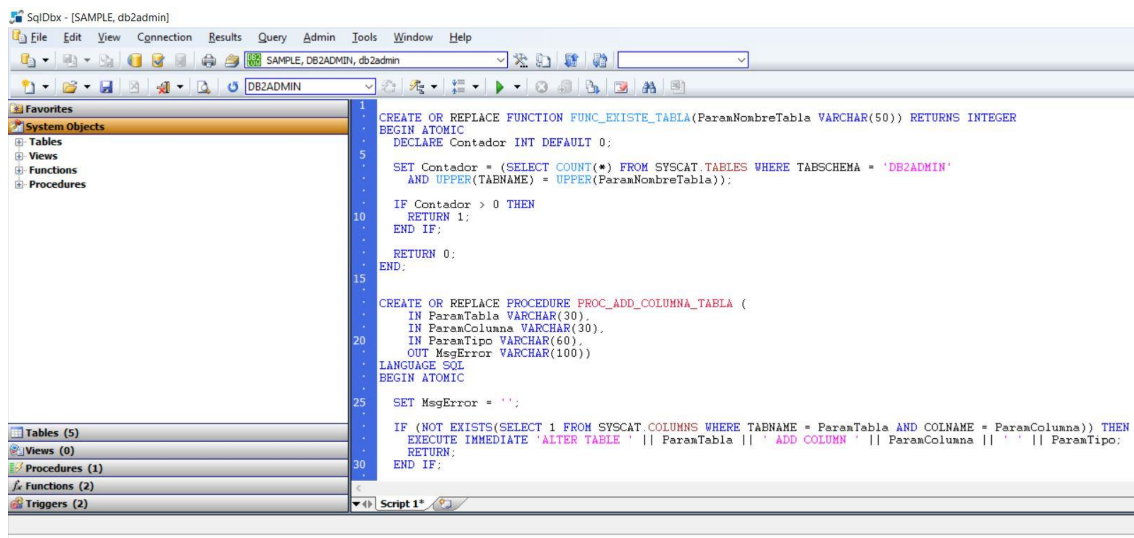

Figura 3. *Ejemplo de interfaz de SqlDbx*

- **DBeaver**

Si la administración que proporciona SqlDbx no cumple con tus expectativas o requisitos, DBeaver es otro IDE de administración y desarrollo de base de datos con un mayor número de funcionalidades. Entre las principales destacan: la capacidad de conectarse a bases de datos remotas, la información tan detallada que se puede conocer de nuestras tablas y la visualización de las relaciones entre entidades. Además de soportar múltiples plataformas y bases de datos a parte de DB2.

- **IBM Data Studio**

Este IDE de programación de base de datos se ha incluido en la memoria gracias a que posee la funcionalidad de depurar la ejecución de procedimientos almacenados. Es por esto por lo que nos permite comprobar dónde falla nuestra ejecución o si realiza el flujo que esperábamos.

- **Hibernate**

Toda aplicación que disponga de una base de datos relacional y necesite convertir sus objetos en datos para almacenarlos debería contar con esta herramienta. Hibernate el framework ORM que usaremos para agilizar la relación entre la aplicación y nuestra base de datos. Su integración con Spring es completa. Despreocupa al programador el establecer cómo se guardará, recuperará o eliminará la información que persistamos [10].

Es muy importante conocer su funcionamiento, ya que permite acceder a los datos y mapear automáticamente las entidades con los objetos, reduce notablemente el código y facilita las consultas y operaciones sobre la base de datos.

- **Spring Data JPA**

Java ofrece un conjunto de reglas y directrices para interactuar con la base de datos. Es independiente del ORM que utilicemos y establece el estándar a seguir para realizar operaciones sobre la base de datos. Esta API proporciona a nuestros repositorios los métodos básicos para administrar y desarrollar nuestras operaciones sobre la capa de persistencia [11].

La diferencia con Hibernate es que éste es una herramienta de mapeo entre objetos y las entidades que forman la base de datos y Spring Data JPA es el API que usa el ORM para persistir los datos. A lo largo de esta memoria veremos cómo implementarlo, qué uso se le da y la importancia de contar con estas tecnologías.

3. Retos principales

Una vez explicado el contexto de las tecnologías que se van a usar, en este apartado se van a plantear y analizar los problemas a los que la mayoría de los desarrolladores backend, que usan Spring, van a tener que enfrentarse.

3.1 Servicio REST con lógica PL/SQL

Para entender los problemas que se van a mencionar y por qué han sido elegidos estos, es necesario explicar el prisma del que partimos.

Bien es cierto que, a lo largo de la carrera, se nos enseña a programar en lenguajes de alto nivel y se puede llegar a dar por sentado que en la vida laboral nos encontraremos ante la misma perspectiva. Es por eso por lo que se debe hacer saber que, muchas empresas poseen código de programas antiguos con muchísima funcionalidad implementada en lenguajes que dejaron de estar de moda hace tiempo. Lenguajes cuyas instrucciones eran más parejas a lenguaje máquina o directamente poseían toda la lógica en la capa de persistencia con el objetivo de que, independientemente del lenguaje que se vaya a usar en el futuro, cierta funcionalidad quedara incrustada y protegida si se mantenía la misma base de datos.

Como consecuencia, el punto de vista, desde el que abordaremos los problemas que se van a plantear, es el de un programador backend cuyas tareas principales serán las de crear una API Rest que acceda a entidades de la base de datos desde un controlador y, por otra parte, llamar a procedimientos almacenados y funciones que estarán en la capa de persistencia programados en PL/SQL.

La arquitectura que seguiremos para crear la API Rest se detallará en el apartado de Diseño de la arquitectura y, más tarde, en el apartado de Desarrollo se analizará el código necesario para su funcionamiento. En este último apartado, también se aprenderá cómo llamar a funciones y procedimientos almacenados implementados en DB2 desde Spring.

Adicionalmente, dentro de este apartado plantearemos cuatro tareas que podrían ser de utilidad a cualquier programador backend. Su análisis específico estará en el apartado de Diseño de la arquitectura y la parte de programación en el apartado de Desarrollo.

3.2 Thymeleaf como motor de plantillas HTML

Aunque en ámbitos empresariales se suela separar el backend y el frontend, y este último se encuentre programado con otros frameworks como, por ejemplo, Angular, Vue o React, también se pueden crear plantillas HTML que reciban datos por parte del servidor y los muestre gracias a la integración de Thymeleaf con Spring.

En el subapartado 5.3 de esta memoria, veremos los atributos más importantes para trabajar con Thymeleaf y poder crear plantillas, destacando sus características principales, las ventajas que nos proporciona y las limitaciones que posee.

3.3 Unificar el enrutamiento con un gateway

Como consecuencia de la facilidad que nos ofrece Spring Boot para crear microservicios, es probable que, a lo largo del tiempo, creemos una cantidad considerable de microservicios que estén conectados de forma poco intuitiva. Por este motivo, si queremos que nuestros compañeros de trabajo sepan cómo se comunican entre ellos, tendremos que gastar tiempo explicando qué hace cada microservicio y qué microservicios reciben datos de cada uno de ellos.

La solución que se propone en el subapartado 4.3 es programar un API Gateway que unifique todos los puntos de enrutamiento en un único proyecto. Para ello, usaremos Spring Cloud Gateway ya que el objetivo es una integración completa de Spring y no usar API externas como podrían ser Kong, 42crunch, Istio, AWS API Gateway, Tyk y 3scale a pesar de que nos puedan ofrecer mejores características.

3.4 Registrar y localizar nuestros microservicios

El módulo de Spring Cloud Discovery nos ofrece la posibilidad añadir las dependencias de Eureka Server y Eureka Discovery Client para registrar información de los microservicios y tenerlos localizados. Eureka Server se añadirá en un microservicio que se encontrará siempre activo, con la finalidad de registrar las instancias que se encuentren activas de otros microservicios a los cuales marcaremos como Eureka Client.

La implementación de Eureka Server se encontrará en el subapartado 5.5 donde se conocerán las anotaciones más importantes que debemos usar. Además, analizaremos la información que podemos consultar desde el *dashboard* que proporciona esta tarea.

3.5 Evitar validaciones de otras entidades

Es muy común añadir ciertas validaciones previas o posteriores al realizar inserciones, borrados o actualizaciones de registros de las entidades de la base de datos. Es decir, comprobar que cierto registro de alguna entidad cumple con unas características que deseamos al guardarla en la base de datos. Por ejemplo, que la fecha de caducidad no sea anterior a la fecha actual a la hora de insertar un producto.

Si llevamos esto al extremo, es probable caer en la cuenta de que, si todas las tablas tienen validaciones, cada vez que insertemos un registro en la base de datos, desde la capa de negocio, tendremos que cumplir con todas las validaciones de otros objetos. Esto supone que cuando hagamos un test para comprobar si cierta validación es ejecutada con éxito, deberemos tener en cuenta las validaciones de otras entidades cuando realmente esas comprobaciones ya estarían en su test correspondiente.

A diferencia del resto de tareas, debido a su extensión, ésta será el contenido del apartado de Pruebas donde veremos cómo desactivar las validaciones del resto de entidades que no queremos tener en cuenta al realizar operaciones sobre la base de datos.

3.6 Automatizar la creación de la estructura de test

Cuando se detalle el apartado anterior quedará bien especificado la estructura que tendremos que seguir para realizar validaciones correctamente. Ahí podremos observar que los ficheros que componen la estructura de test es siempre la misma y solo cambian los datos que contienen.

Por este motivo, en el apartado de Trabajos futuros se planteará una alternativa para automatizar la creación de estos ficheros. La decisión de implementarlo en este apartado es debida a que ya existe mucho contenido en los otros apartados, pero se considera importante conocer qué son las tareas Gradle que nos permiten automatizar este tipo de ideas.

Independientemente de lo planteado en este apartado, lo relevante de esta solución es la cantidad de opciones que nos ofrece Gradle para automatizar código. Es decir, una vez visto el ejemplo que presentaremos, el programador conocerá una herramienta idónea para poder crear su propias tareas Gradle que están integradas perfectamente con Spring. A su vez, son una nueva forma de realizar scripts que se podrán ejecutar desde la línea de comandos, también conocidos como CLI, y que pueden servir para satisfacer innumerables ideas dentro del proyecto.

4. Diseño de la arquitectura

Mencionadas las tecnologías y planteados los problemas a los que los programadores backend se pueden llegar a enfrentar, es hora de explicar cómo es la arquitectura que van a seguir las soluciones.

4.1 Base de datos

En el apartado de Estado del arte hemos hablado de la base de datos DB2. Aunque su uso no es tan popular en proyectos a pequeña escala, sí lo es en ámbitos empresariales. DB2 utiliza SQL como lenguaje de consulta, pero también soporta PL/SQL que amplia SQL con elementos característicos de los lenguajes de programación: declaración de variables, bucles, control de flujo, funciones…

Por este motivo, dado que existe la posibilidad de tener lógica en esta capa y nos pueden mandar crear funciones o procedimientos almacenados, se considera importante conocer su declaración para luego poder trabajar con ellos en Spring.

FUNCIONES

Las funciones DB2 reciben parámetros separados por comas y únicamente devuelven un resultado. Se recomienda usar un prefijo para diferenciar las creadas por el programador de las predefinidas por DB2, en nuestro caso será “FUNC_”. A su vez, es aconsejable que los parámetros tengan el prefijo “Param” para diferenciarlos de las variables que se pudieran crear dentro de ella. A parte, tanto los parámetros como las variables que se declaren dentro deberían tener un valor por defecto. Por último, cabe mencionar que las funciones pueden ser atómicas. Es decir, si lo son y fallan, el estado de la base de datos permanece como antes de haber llamado a esa función. Además, al ser atómicas acceden a un mismo estado de los datos de la base de datos y evitan inconsistencias.

Sintaxis

```
CREATE OR REPLACE FUNCTION FUNC_EXISTE_TABLA(ParamNombreTabla VARCHAR(50)) RETURNS INTEGER
BEGIN ATOMIC
  DECLARE Contador INT DEFAULT 0;

  SET Contador = (SELECT COUNT(*) FROM SYSCAT.TABLES WHERE TABSCHEMA = 'DB2ADMIN'
    AND UPPER(TABNAME) = UPPER(ParamNombreTabla));

  IF Contador > 0 THEN
    RETURN 1;
  END IF;

  RETURN 0;
END;
```

Figura 4. *Ejemplo de una función en DB2*

Ejecución

La función del ejemplo realiza una operación de búsqueda en la vista del catálogo del sistema para saber si existe la tabla que le pasemos por parámetro. En caso de que la tabla exista el resultado de la ejecución sería un uno, sino un cero.

La forma que utilizaremos para llamar a las funciones se apoyará en una tabla virtual o fantasma que posee DB2 llamada SYSIBM.SYSDUMMY1:

```
SELECT FUNC_EXISTE_TABLA('TABLA_EJEMPLO') FROM SYSIBM.SYSDUMMY1;
```

PROCEDIMIENTOS ALMACENADOS

Los procedimientos almacenado de DB2, a diferencia de las funciones, pueden recibir múltiples parámetros separados por comas y devolver más de un resultado o ninguno. En los procedimientos almacenados mantendremos las normas de prefijar los parámetros con "Param" y asignar valores por defecto con el mismo objetivo que en las funciones. También pueden ser atómicos. Se ejecutarán correctamente todas las instrucciones o si hay alguna excepción se llevará a cabo un *rollback* y ningún cambio será aplicado.

Continuando con la explicación, se ha de comentar que los procedimientos almacenado se utilizan debido a que el código ejecutado siempre es más rápido que si hiciésemos el mismo proceso desde Spring. Por si fuera poco, toda la lógica que se escriba mediante procedimientos en DB2 estará integrada de forma definitiva independientemente del framework que lo ejecute por las capas superiores, a no ser que cambie de base de datos. También permite añadir una capa de seguridad ya que es más difícil acceder a estos procesos para el usuario.

Para concluir, es importante saber que dentro de un procedimiento almacenado se pueden llamar funciones, pero no al revés.

Sintaxis

```
CREATE OR REPLACE PROCEDURE PROC_ADD_COLUMNA_TABLA (
    IN ParamTabla VARCHAR(30),
    IN ParamColumna VARCHAR(30),
    IN ParamTipo VARCHAR(60),
    OUT MsgError VARCHAR(100))
LANGUAGE SQL
BEGIN ATOMIC

  SET MsgError = '';

  IF (NOT EXISTS(SELECT 1 FROM SYSCAT.COLUMNS WHERE TABNAME = ParamTabla AND COLNAME = ParamColumna)) THEN
    EXECUTE IMMEDIATE 'ALTER TABLE ' || ParamTabla || ' ADD COLUMN ' || ParamColumna || ' ' || ParamTipo;
    RETURN;
  END IF;

  SET MsgError = 'No se ha realizado la operacion';
END;
```

Figura 5. *Ejemplo de procedimiento almacenado en DB2*

Ejecución

A diferencia de la ejecución de las funciones, no se necesita un SELECT para lanzar un procedimientos almacenado. Utilizamos la palabra reservada "CALL" acompañada del nombre del procedimiento con sus parámetros rellenos salvo los de salida.

```
CALL PROC_ADD_COLUMNA_TABLA('TABLA_EJEMPLO', 'PRUEBA', 'INTEGER
DEFAULT 0', ?MSGERROR$VARCHAR$OUT);
```

El procedimiento del ejemplo comprueba que no exista una columna con el nombre que le indicamos en la tabla que queremos y en caso de no existir, lo añade. Si existe, devuelve un mensaje de error. Esto es lo más importante de los procedimientos. Es decir, el uso común que se le da es el de realizar procesos que avisen al usuario o al programador en qué ha fallado o si ha tenido éxito. Principalmente su estructura suele ser la siguiente: en primer lugar, declaración de variables y seguidamente comprobar que los datos introducidos son válidos En caso de que no lo sean, la ejecución no debería continuar. Más tarde, operaciones de cálculo o consultas y finalmente, devolver el resultado para conocer el estado de la ejecución.

4.2 Patrón MVC e inyección de dependencias

La tarea principal de esta memoria es crear una API Rest con Spring debido a la demanda y la utilidad que posee. Para que su comprensión sea completa, tenemos que seguir allanando el camino antes de meternos de lleno con código Spring. Por este motivo, antes de explicar cómo crear una API Rest necesitamos conocer las características básicas del patrón MVC y la propiedad más relevante de Spring para este proyecto conocida como inyección de dependencias.

PATRÓN MVC

Este patrón de arquitectura software no es nuevo. Su origen ya tiene unas décadas, pero su popularidad se ha incrementado en los últimos años ya que numerosos frameworks lo utilizan como patrón de arquitectura para desarrollo web.

El pilar fundamental en el que se apoya es la separación del código en tres capas: [12] [13]

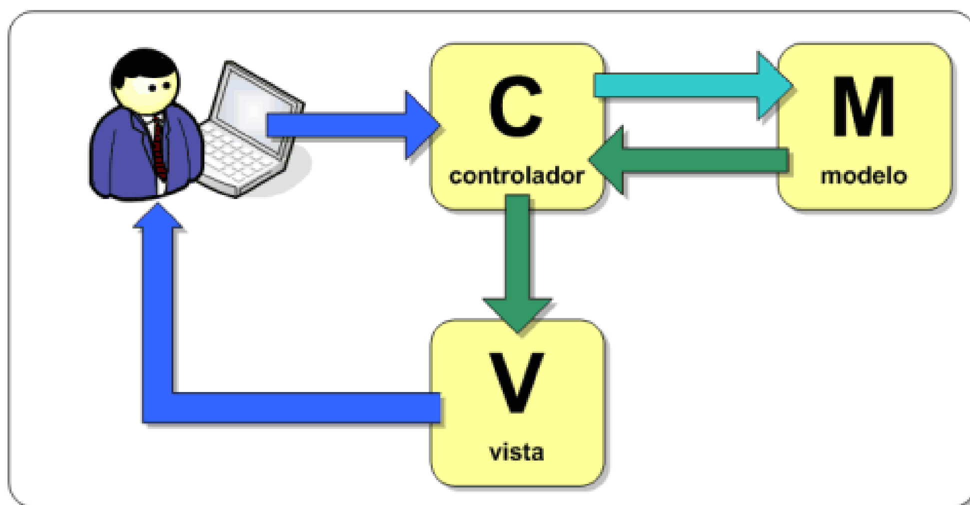

Figura 6. *Esquema patrón MVC*

El modelo

Es el conjunto de la representación de los datos de la aplicación, la lógica del sistema y los mecanismos de persistencia. Idealmente, esta capa debe ser independiente al sistema de almacenamiento.

En Spring, diferenciaremos entre modelos y entidades. Los modelos serán los objetos que enviemos y recibamos del controlador y las entidades serán las que usará nuestro ORM para realizar consultas y operaciones en la base de datos. Para trabajar con ambos en un mismo camino, necesitaremos de clases (*converters*) que conviertan de modelo a entidad y a la inversa. Aunque no es la única forma que veremos.

El controlador

Hace referencia al intermediario entre el modelo y la vista. Es el encargado de recibir peticiones por parte del cliente y enviar datos a las vistas. El código de esta capa no debe manipular los datos, ni mostrar ningún tipo de salida. Su responsabilidad es servir de enlace recibiendo los eventos o acciones que la aplicación posee.

La vista

Como su nombre indica, esta capa presenta los datos al usuario para que interactúe con ellos. Aunque en esta capa se trabaje con los datos, no se interactúa directamente con ellos. Recibe los modelos que el controlador le envía y muestra la salida, generalmente, usando código HTML.

Flujo patrón MVC en Spring

1. El usuario, mediante cualquier tipo de acción, interactúa con algún elemento de la interfaz.
2. El controlador gestiona el evento recibiendo los datos y encapsulándolos en algún modelo esperado.
3. El controlar envía el modelo a algún servicio de la capa de negocio donde se realizarán modificaciones en los datos.
4. El servicio que manipula esos datos tendrá acceso a la capa de persistencia gracias al ORM y podrá insertar, actualizar, eliminar o consultar los datos. Previamente a esto, habrá convertido el modelo en una entidad.
5. Una vez realizados las operaciones en la capa de persistencia, si eran necesarios, el servicio convierte la entidad en un modelo que retornará al controlador.
6. El controlador recibirá el nuevo modelo y lo enviará a la vista.
7. La vista actualizará la interfaz del usuario con los datos manipulados correctamente o mostrará algún mensaje de error en caso de haber encontrado alguna excepción. De esta forma, el usuario decidirá si repetir el proceso o realizar uno nuevo.

Ventajas

- La independencia entre capas permite que el sistema sea más robusto, flexible, limpio y mantenible. A su vez, admite tener varias vistas con un mismo modelo.
- Permite un mejor desarrollo del entorno de trabajo en equipo. Cada programador puede especializarse en una capa y trabajar en paralelo.
- Este patrón facilita la creación de pruebas unitarias que favorezcan tener todo nuestro código protegido ante fallos.

INYECCIÓN DE DEPENDENCIAS

Hasta ahora habíamos visto que si una clase necesita un objeto debe instanciarlo para poder usarlo. Mediante la inyección de dependencias quien realiza ese proceso e inyecta el objeto es Spring. Dicho en otras palabras, este patrón de diseño suministra objetos a una clase en lugar de ser la propia clase quien cree dichos objetos. Esto es posible ya que Spring posee un contenedor en el que se encuentran todos los objetos que hemos marcado mediante anotaciones, que luego se detallarán, para que gestione la inyección de estos. Una vez marcados, se añaden a ese contenedor y quedan accesibles para el resto del programa. Si se necesita, Spring inyecta la dependencia hasta que se deje de hacer uso.

Esta característica tan destacable de Spring ofrece un desacoplamiento entre la creación de objetos y sus llamadas. Es decir, la dependencia sigue siendo accesible aunque la clase se modifique y a la inversa.

Para insertar una clase en el contenedor debemos añadir la anotación genérica @Component. Aunque en el apartado de desarrollo veremos las anotaciones con mayor profundidad, destacaremos las anotaciones @RestController, @Service y @Repository para satisfacer lo mencionado del flujo del patrón MVC.

La anotación @RestController cumplirá el estereotipo del controlador que recibe datos de la vista y delega el tratamiento de los datos a la capa de modelo donde se encuentran los @Service. Los servicios implementan una interfaz donde se encuentran los métodos que deben sobrescribir con la lógica. A su vez, los @Service tienen acceso a los @Repository. Los repositorios se declaran como interfaz que extiende de JpaRepository para acceder a los registros de la base de datos.

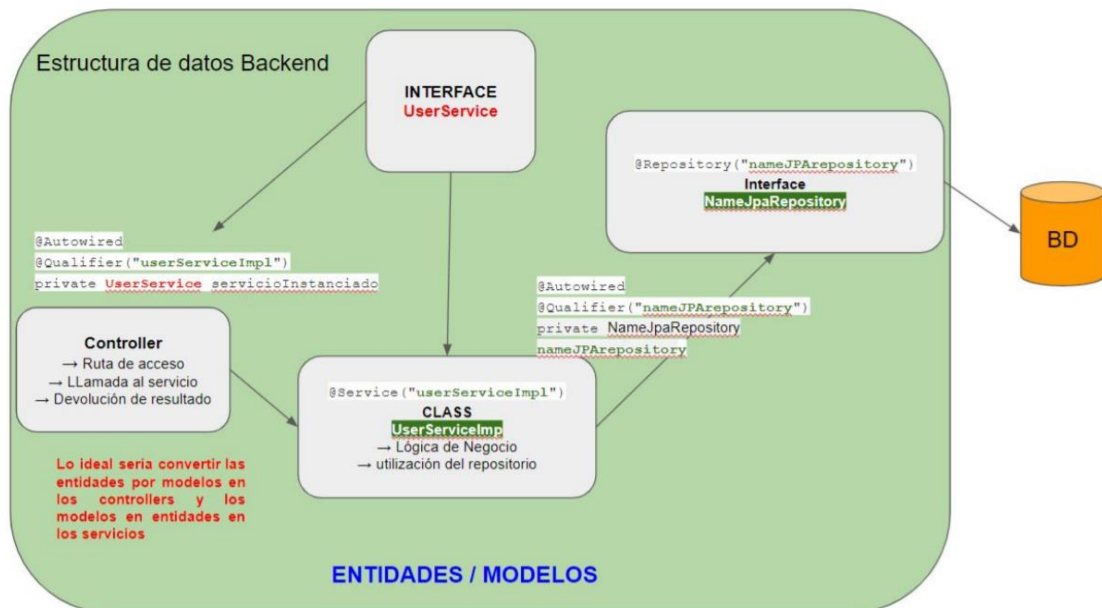

Figura 7. *Esquema de la inyección de dependencias en Spring*

Para finalizar con este subapartado, se debe saber que el tratamiento de las dependencias de los objetos de Spring sigue el patrón *singleton*. Esto significa que en el contenedor solo existe una instancia de cada objeto y, por tanto, todas las clases acceden al mismo objeto cuando se solicita una instancia. [13]

Para cambiar el alcance de las dependencias tendremos que hacer uso de la anotación @Scope. Las opciones que nos ofrece Spring son las siguientes: [13] [14]

- @Scope("*singleton*"): Es el por defecto, se crea una instancia del objeto para toda la aplicación.
- @Scope("*prototype*"): Una instancia nueva cada vez que se hace uso del objeto.
- @Scope("*request*"): Una instancia nueva por cada petición HTTP request.
- @Scope("*session*"): Una instancia nueva por sesión HTTP.
- @Scope("*globalSession*"): Una instancia nueva por sesión HTTP global.

4.3 API Gateway

Como se mencionó en el apartado de Retos principales es muy común tener microservicios interconectados entre ellos. Esto no supone ningún problema para un número pequeño de microservicios, pero sí cuando se supera cierto límite. La arquitectura que presentaría esta situación es la siguiente: [15]

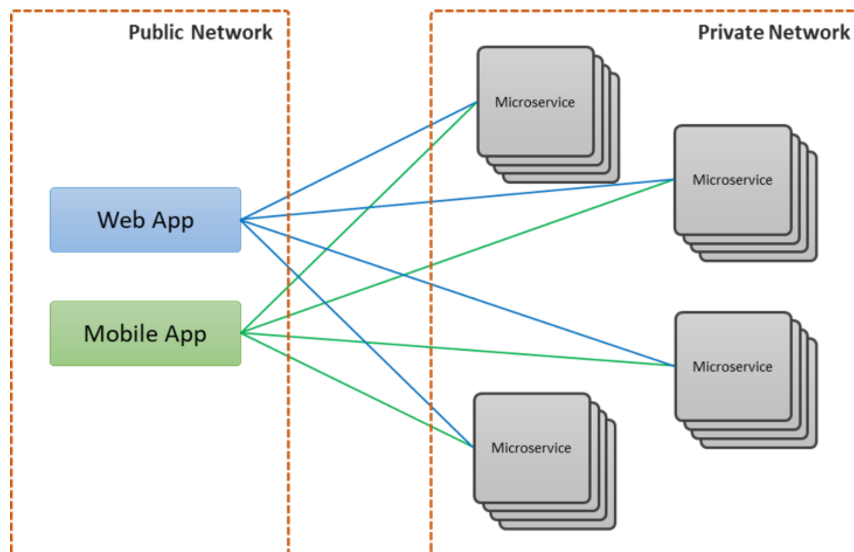

Figura 8. *Arquitectura microservicios sin API Gateway*

Las desventajas principales en ámbitos laborales son el tener que explicar a cada empleado qué microservicios se conectan con el resto y la localización de sus puertos. Ya que no todos los microservicios tienen por qué conectarse con los demás.

Como solución a este inconveniente, se plantea usar una API Gateway para unificar el enrutamiento:

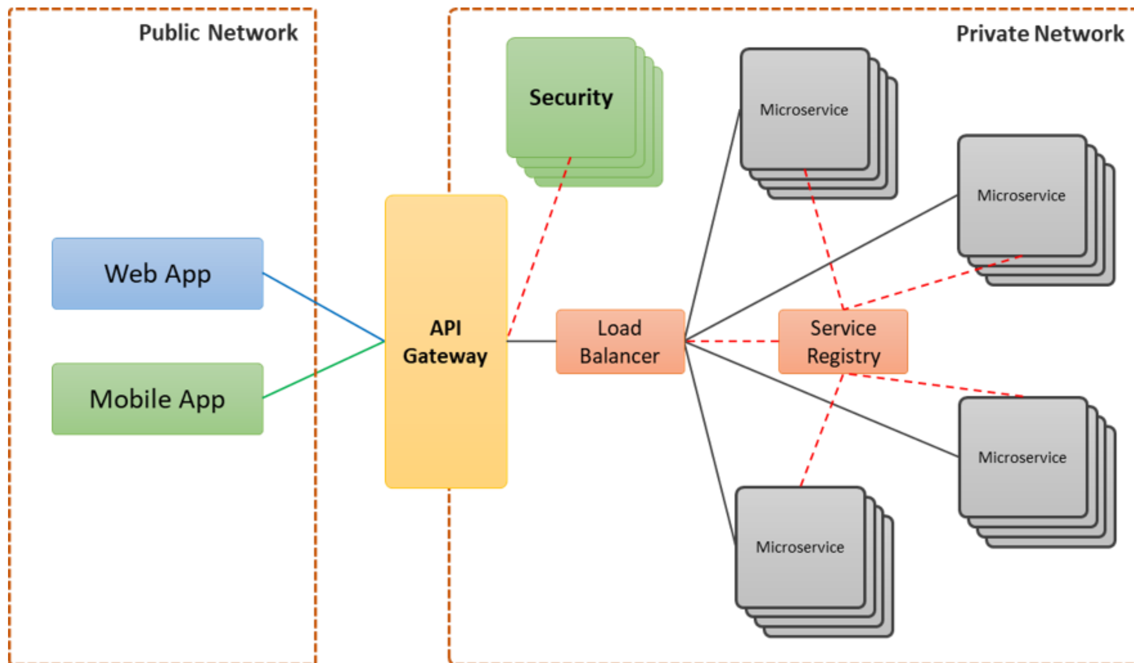

Figura 9. *Arquitectura microservicios con API Gateway*

Aunque el número de ventajas sea superior al número de inconvenientes cabe destacar cuáles son los problemas que puede suponer la implementación de esta tarea:

- Es un proyecto más a configurar por los programadores que trabajen en el desarrollo. A cada microservicio nuevo que se añada se debe tener en cuenta y añadir su ruta dentro del proyecto del API Gateway.
- Como cualquier otro proyecto, éste también puede fallar sino se configura como es debido. Además, a largo plazo hay que cuidar que no se convierta en una caja negra de código que solo escala sin ser refactorizado.
- Una vez implementado con un correcto funcionamiento, se debe reunir a los trabajadores del equipo y explicar cómo funciona y qué ventajas y desventajas conlleva. Sobre todo para los programadores frontend que deben cambiar las rutas API para que apunten a la nueva incorporación.
- Al ocupar la posición que tiene, este tipo de implementación añade una latencia extra que se debe considerar.
- Por último, la incorporación de la API Gateway va en contra de la idea central de trabajar con microservicios. Esto es debido a que añade cierta dependencia entre microservicios y elimina su autonomía.

5. Desarrollo

En este apartado se mostrará el código necesario para implementar las propuestas mencionadas. El objetivo principal es que el programador entienda cuáles son las anotaciones principales de Spring, qué uso se le dan y las peculiaridades de cada tarea.

Como consecuencia de que esta memoria trata de ser una guía de utilidad para el aprendizaje de Spring y pretende plasmar todo el conocimiento sin ocultar pasos, se advierte de que habrá más código de lo recomendado en este apartado.

5.1 API Rest con Spring

La finalidad de la API Rest que vamos a construir será la de consultar e insertar datos en la base de datos. Para conseguirlo, tendremos que seguir la siguiente estructura:

En primer lugar, crearemos la super clase que definirá las columnas de la entidad a persistir. A continuación, la entidad extendida. El objetivo de la entidad extendida es añadir funcionalidad extra a la super clase y que el código sea más limpio. Por ejemplo, añadir las relaciones entre tablas, sobrescribir métodos que guarden fechas con el formato que deseemos, llamar a la clase que tiene las validaciones que más tarde analizaremos, etc. Adicionalmente, necesitaremos crear el repositorio que permita el enlace entre la base de datos y la capa de negocio. También, crearemos un servicio que será quien inyecte la dependencia del repositorio para trabajar con él en función de las necesidades. Por último, en el controlador añadiremos una ruta GET y otra POST para consultar y añadir registros en la base de datos. Este controlador trabajará con el modelo de la entidad que también crearemos.

El esquema que seguiremos para trabajar con las entidades y los repositorios se apoyará en el ORM usando JPA de la siguiente forma:

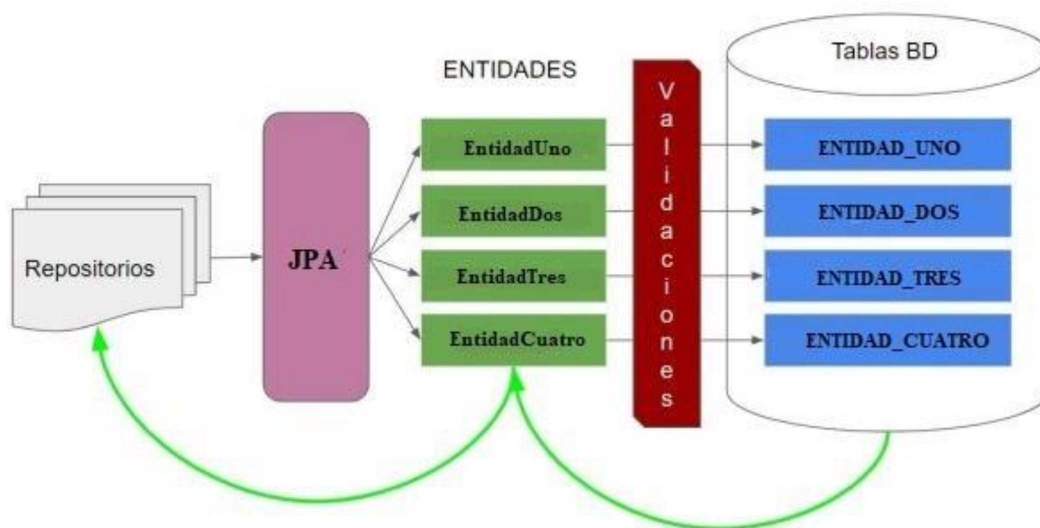

Figura 10. *Uso del ORM en Spring*

Para satisfacer el esquema, lo primero que haremos será crear la propia entidad. Ésta estará compuesta por la propia entidad y una super clase de la cual heredamos sus columnas. Esta herencia es posible gracias a Hibernate y será la forma en la que podremos añadir la entidad extendida:

SUPER CLASE

En primer lugar, tendremos que crear un paquete donde guardemos todas nuestras super clases. Se recomienda al programador investigar más sobre los atributos que aceptan las anotaciones debido a que se van a mencionar los necesarios para su implementación. Para un mejor entendimiento, analizaremos el proceso en función a un código ya implementado y así explicar el uso de las anotaciones principales que usaremos para implementar la super clase:

```
@MappedSuperclass
@Table( name ="CFG_TEST")
public class CfgTest implements Serializable {  Complexity is 3 Everything is cool!

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "ID", nullable = false, length = 4)
    protected Integer id;

    @Column(name = "FICHERO", length = 250)
    protected String fichero;

    public CfgTest() {
    }
```

Figura 11. *Implementación de una super clase*

- @MappedSuperClass es la principal. Indica que esta clase no será la entidad, pero las que hereden de ella heredarán sus columnas.
- @Table indica el nombre de la tabla que mapeará la entidad. También se puede acompañar del esquema en el que se encuentra la tabla.
- La clase implementa la interfaz Serializable para poder ser enviada a través de la red.
- @Id indica que esa columna es la clave primaria. El caso donde la clave primaria está compuesta por más de una columna se verá más tarde.
- @GeneratedValue se acompaña de la estrategia que deseemos. En este caso, indicamos que Hibernate autoincremente su valor en cada inserción.
- @Basic marcado con opcional a falso permite que a nivel de Java se requiera ese campo para ser insertado.
- @Column se acompaña del nombre de la columna, la propiedad NOT NULL en la base de datos y la longitud del campo.

Es importante que nuestra super clase tenga el constructor vacío, los getters y los setters.

Para crear una clave primaria compuesta por más de una columna se recomienda crear una clase que represente esas columnas de la siguiente forma:

```
@Embeddable
public class CfgTestOpcionPK implements Serializable {

    @Basic(optional = false)
    @Column(name = "ID_TEST", nullable = false)
    protected Integer idTest;
    @Basic(optional = false)
    @Column(name = "ID_OPCION", nullable = false)
    protected Integer idOpcion;
    public CfgTestOpcionPK() {
    }
```

Figura 12. *Clave primaria compuesta*

```
@MappedSuperclass
@Table( name ="CFG_TEST_OPCION")
public class CfgTestOpcion implements Serializable {

    @EmbeddedId
    protected CfgTestOpcionPK cfgTestOpcionPK;

    public CfgTestOpcion() {
    }
```

Figura 13. *Inyección de la clave primaria compuesta*

De esta forma, con la anotación @Embeddable marcamos la clase como inyectable y mediante @EmbeddedId inyectamos esa misma clase.

ENTIDAD

La entidad o entidad extendida será la clase con la que realmente trabajemos, ya que tiene las columnas de la clase que hereda y la funcionalidad que añadamos según nuestras necesidades. Al igual que con las super clases, se recomienda crear un paquete dentro del proyecto donde guardar todas nuestras entidades.

La principal diferencia es que esta clase si estará marcada con la anotación @Entity y tendrá que extender de la super clase e implementar Serializable:

```
@Entity
@Table(name = "TABLA_EJEMPLO")
public class TablaEjemploExtends extends TablaEjemplo implements Serializable { ... }
```

Figura 14. *Entidad extendida de ejemplo*

El punto más importante que vamos a desarrollar de la lógica que se puede añadir en las clases extendidas son las relaciones Hibernate entre entidades: [16]

@OneToOne

```
@OneToOne(cascade = CascadeType.ALL, mappedBy = "nombreAtributo", fetch = FetchType.LAZY)
private ClaseRelacionada nombreClaseRelacionada;
```

Figura 15. *Relación @OneToOne*

La anotación @OneToOne tiene los siguientes atributos:

- Cascade: es tarea del programador acertar en el uso de este atributo ya que no hay una regla universal. En este caso se ha usado *CascadeType.ALL* para darle legibilidad del código y para que se actualice la tabla destino ante cualquier operación. No se recomienda usar como comodín.
- MappedBy: entre comillas se debe insertar el nombre del atributo que deseemos relacionar en la tabla destino.
- Fetch: no carga la relación a menos que se invoque el getter. Para realizar lo contrario, marcar como *EAGER*.

@OneToMany

A diferencia de la anterior, si se marca una relación como @OneToMany debe existir otra que sea @ManyToOne.

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "test", fetch = FetchType.LAZY)
protected List<CfgTestOpcionExtends> cfgTestOpcionExtendsList;
```

Figura 16. *Relación @OneToMany*

Idéntica a la anterior salvo porque declara una lista como atributo. Es obligatorio el *mappedBy*.

```
@JoinColumn(name = "ID_TEST", referencedColumnName = "ID", nullable = false, insertable = false, updatable = false)
@ManyToOne(optional = false, fetch = FetchType.LAZY)
protected CfgTestExtends test;
```

Figura 17. *Relación @ManyToOne para satisfacer @OneToMany*

A destacar, la anotación @JoinColumn. Puede confundirse con @Column, pero esta anotación nos permite definir el nombre exacto de la columna, si debe ser insertable, actualizable y nulo. En este caso, el atributo *name* hace referencia al nombre que deseemos que tenga la columna de esa clase, pero el atributo *referencedColumnName* es el nombre que le hemos puesto en la clase destino.

@ManyToOne

Realmente ya estaría explicada con lo mencionada anteriormente, pero se va a profundizar en el caso de que la relación una dos columnas:

```
@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumns({@JoinColumn(name = "COLUMNA_UNO"), @JoinColumn(name = "COLUMNA_DOS")})
private ClaseRelacionada nombreClaseRelacionada;
```

Figura 18. *Múltiples columnas en una unión*

Gracias a la anotación @JoinColumns podremos crear múltiples columnas de la unión.

@ManyToMany

Para entender este caso, vamos a plantear la situación en la que un usuario puede tener diversos roles y los roles pueden estar asignados a más de un usuario. La forma en la que podemos relacionar ambas entidades es la siguiente:

En la clase Usuarios:

```
@JoinTable(name = "ROL_USUARIOS", joinColumns = {
        @JoinColumn(name = "USUARIO", referencedColumnName = "USUARIO", nullable = false)},
        inverseJoinColumns = {
        @JoinColumn(name = "ID_ROL", referencedColumnName = "ID", nullable = false)})
@ManyToMany(fetch = FetchType.LAZY)
private List<RolesExtends> rolesList;
```

Figura 19. *Relación @ManyToMany en usuarios*

La forma que en que hemos orientado la solución es la de crear una tercera tabla adicional que reúna una columna de cada tabla. La anotación @JoinTable realiza dicha acción en la nueva tabla ROL_USUARIOS uniendo el identificador del usuario y el identificador del rol. Se usa *inverseJoinColumns* para establecer la columna que no pertenece a la clase donde se implementa. En este caso, se llama a la ajena Roles desde la clase Usuarios.

En la clase Roles:

```
@ManyToMany(mappedBy = "rolesList", fetch = FetchType.LAZY)
private List<UsuariosExtends> usuarisList;
```

Figura 20. *Relación @ManyToMany en roles*

Dado que se ha realizado el proceso principal en la clase Usuarios, en ésta solo necesitamos indicar en el *mappedBy* el nombre del atributo al cual hace referencia.

MODELOS

Las clases que se encuentren dentro del paquete de *models* serán las que el controlador utilice para enviar datos a la vista o recibirlos. El modelo de una entidad se asemeja en su totalidad salvo que no debe tener ninguna anotación Spring y, en algunos casos, no se debe incluir la clave primaria ya que es un valor autoincrementable que Hibernate maneja por su cuenta. Tampoco suelen extender de ninguna clase a no ser que se necesite incrementar un modelo ya existente. Hay que destacar que en su contenido deben figurar los constructores vacíos, los getters y los setters.

Un ejemplo de modelo sería el siguiente:

```
public class CfgTestModel {

    private int id;
    private String fichero;

    public CfgTestModel() {
    }

    public CfgTestModel(int id, String fichero) {
        this.id = id;
        this.fichero = fichero;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getFichero() {
        return fichero;
    }

    public void setFichero(String fichero) {
        this.fichero = fichero;
    }
}
```

Figura 21. *Modelo de ejemplo*

CONVERTERS

Como se ha mencionado previamente, los controladores envían y reciben modelos y los servicios trabajan con entidades. Como consecuencia de esto, necesitamos alguna forma de cambiar de modelo a entidad y a la inversa. Aunque esta práctica no es la más popular, sí es útil y cumple nuestro objetivo.

La solución que se propone es la de añadir un componente que mediante dos métodos satisfaga esta necesidad. El primero convertiría de entidad a modelo y el segundo realizaría el proceso inverso:

```
@Component("alumnoConverter")
public class AlumnoConverter {

    public AlumnoModel alumno2model(Alumno alumno) {
        AlumnoModel alumnoModel = new AlumnoModel();
        alumnoModel.setDni(alumno.getDniAlumno());
        alumnoModel.setNombre(alumno.getNombre());
        alumnoModel.setApellido(alumno.getApellido());
        alumnoModel.setEdad(alumno.getEdad());
        alumnoModel.setEmail(alumno.getEmail());
        alumnoModel.setTelefono(alumno.getTelefono());
        alumnoModel.setCiudad(alumno.getCiudad());
        alumnoModel.setPais(alumno.getPais());
        return alumnoModel;
    }

    public Alumno model2alumno(AlumnoModel alumnoModel) {
        Alumno alumno = new Alumno();
        alumno.setDniAlumno(alumnoModel.getDni());
        alumno.setNombre(alumnoModel.getNombre());
        alumno.setApellido(alumnoModel.getApellido());
        alumno.setEdad(alumnoModel.getEdad());
        alumno.setEmail(alumnoModel.getEmail());
        alumno.setTelefono(alumnoModel.getTelefono());
        alumno.setCiudad(alumnoModel.getCiudad());
        alumno.setPais(alumnoModel.getPais());
        return alumno;
    }

}
```

Figura 22. *Converter de ejemplo*

Dado que esta solución puede tener numerosas alternativas, en el siguiente apartado analizaremos los repositorios y destacaremos una posibilidad de realizar este proceso de forma automática gracias a Hibernate.

REPOSITORIOS

Para poder acceder a la capa de persistencia utilizaremos los repositorios. Estos se apoyarán Spring Data JPA que nos ofrece las reglas y directrices necesarias para interactuar con la base de datos.

La forma más sencilla de acceder a los datos se implementa considerando a los repositorios como interfaces y haciendo que extienda de JpaRepository. Previamente tendremos que haber marcado la interfaz con la anotación @Repository. En caso de que no se añada un nombre al lado de la anotación, el identificador será el nombre de la interfaz con la primera inicial en minúscula.

JpaRepository solo nos pedirá la entidad a la cual queremos tener acceso y el tipo de su clave primaria. En caso de que la clave primaria sea compuesta se debe poner la clase que contiene la anotación @Embeddable.

```
@Repository("cfgTestRepository")
public interface CfgTestRepository extends JpaRepository<CfgTestExtends, Integer> {

}
```

Figura 23. *Repositorio de ejemplo*

Hay tres convenciones principales que usaremos para acceder a los datos:

1. Usando los métodos que ya ofrece JPA.

```
<S extends T> S save(S entity);

Optional<T> findById(ID primaryKey);

Iterable<T> findAll();

long count();

void delete(T entity);

void deleteById(ID primaryKey);

void deleteAll();

boolean existsById(ID primaryKey);
```

2. Mediante consultas personalizadas.

```
@Query(value = "select c from CfgTestExtends c where c.fichero=:fichero")
Optional<CfgTestExtends> getByFichero(@Param("fichero") String fichero);
```

Figura 24. *Consulta SQL desde Java para acceder a los datos*

Devuelve un *Optional* obligando al programador decidir qué hacer en caso de que la consulta devuelva null. Para pasar parámetros se usa la anotación @Param acompañado entrecomillas del nombre que tendrá el parámetro dentro de la consulta.

La variedad de opciones que se ofrecen con esta alternativa es incontable. Por este motivo, es la más popular y la que se recomienda dominar.

3. Creando métodos usando palabras reservadas.

Por si fuera poco, JPA permite crear métodos que no requieren de consultas y accedan a los datos utilizando las palabras reservadas que nos facilitan el acceso. Para destacar la importancia y la ventaja que supone utilizar Spring Data JPA se deja adjunta la JPQL *snippet* completa que se puede encontrar en la documentación: [17]

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = 1?
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

Figura 25. *JPQL snippet*

Si recordamos, se mencionó que se mostraría una forma de automatizar la conversión de entidad a modelo mediante una alternativa que nos ofrece Hibernate. Para ello necesitamos conocer dos peculiaridades: el path relativo de la entidad que se quiera convertir y haber construido un constructor que admita parámetros. El resultado quedaría así:

```
@Query(value = "select new /*path_relativa*/.cfg_test.CfgTestModel(c.id, c.fichero)
    from CfgTestExtends c")
List<CfgTestModel> getAllTest();
```

Figura 26. *Conversor automático entidad-modelo*

Si sustituimos la parte donde se encuentra el comentario con el path relativo, el método nos devuelve una lista rellena del modelo en vez de con la entidad.

SERVICIOS

Los servicios contendrán la lógica de la aplicación. Trabajarán con las entidades e inyectarán las dependencias de los repositorios, los *converters* y otros servicios que puedan necesitar.

La inyección de dependencias se realiza con la ayuda de las siguientes anotaciones:

```
@Autowired
@Qualifier("cfgTestRepository")
private CfgTestRepository cfgTestRepository;
```

Figura 27. *Ejemplo de inyección de dependencias de un repositorio en un servicio*

@Autowired

Esta anotación es la más habitual y la que todo programador que trabaje con Spring termine escribiendo más veces. Su uso es el de crear la relación entre la clase donde se escribe y la clase que marcamos con esta anotación. Es decir, inyecta la dependencia automáticamente.

@Qualifier

Dado que no queremos cometer errores y teniendo en cuenta que puede haber más de servicio que implemente la misma interfaz, necesitamos añadir la anotación @Qualifier para distinguirlos. Por brevedad no se profundizará más en este tema, pero se recomienda a todo programador que utilice estas anotaciones siempre acompañadas. La detección de errores será más rápida que si no se usa.

Para trabajar con ellos correctamente, deberán implementar una interfaz donde se encuentren los métodos que debe sobrescribir ese servicio. Si la interfaz tiene el nombre *EjemploService*, la implementación debería llamarse *EjemploServiceImpl*.

```
@Service("cfgTestOpcionServiceImpl")
public class CfgTestOpcionServiceImpl implements CfgTestOpcionService
{

     @Autowired
     @Qualifier("cfgTestRepository")
     private CfgTestRepository cfgTestRepository;

     @Override
     public void deleteTest(int id) throws Exception {
        this.cfgTestRepository.deleteById(id);
     }

     @Override
     public List<CfgTestModel> getAllTest() throws Exception {
         return this.cfgTestRepository.getAllTest();
     }
}
```

Figura 28. *Servicio de ejemplo*

CONTROLADORES

Los controladores se encuentran en la última etapa antes de llegar a la vista. Envían y reciben datos mediante modelos e inyectan las dependencias de los servicios para que apliquen la lógica de la aplicación.

Cada controlador posee un ruta base a partir de la cual trabajan las demás. Sino se escribe ninguna por defecto será "/". Esto se traduce en que los métodos que tengan los controladores deben tener, a su vez, subrutas como ahora veremos.

Para indicar que una clase debe satisfacer el estereotipo de un controlador utilizaremos la etiqueta @RestController. Seguidamente, estableceremos la ruta base del controlador en la etiqueta @RequestMapping.

```
@RestController
@RequestMapping("/test")
public class CfgTestController
{
      @Autowired
      @Qualifier("cfgTestOpcionServiceImpl")
      private CfgTestOpcionService cfgTestOpcionService;

      …
}
```

Figura 29. *Controlador de ejemplo*

La base de todo controlador son las peticiones GET, POST, DELETE y PUT. Spring nos ofrece varias alternativas para implementarlas. La más sencilla y popular es mediante las anotaciones @GetMapping, @PostMapping, @DeleteMapping y @PutMapping.

```
@GetMapping(value = "/getAllTest", consumes = MediaType.APPLICATION_JSON_VALUE)
public List<CfgTestModel> getAllTest() throws Exception {
    return this.cfgTestOpcionService.getAllTest();
}

@PostMapping(value = "/deleteTest/{id}")
public void deleteTest(@PathVariable("id") int idTest) throws Exception {
    this.cfgTestOpcionService.deleteTest(idTest));
}
```

Figura 30. *Métodos GET y POST de ejemplo de un controller*

En el atributo *value* establecemos el nombre de la subruta que mencionábamos con anterioridad. En caso de que se nos solicite entregar, por parte del frontend, los datos en formato JSON tendremos que rellenar el atributo *consumes* con el tipo correspondiente.

Para añadir parámetros variables a la ruta utilizaremos "/{nombreEjemploParamVariable}" y en el parámetro del método usaremos la anotación @PathVariable entrecomillando el nombre.

Si ejecutamos la siguiente petición:

```
GET http://localhost:8080/test/getAllTest
Content-Type: application/json
```

Figura 31. *Petición GET*

El resultado que retornaría es el siguiente:

```
[
  {
    "id": 4,
    "fichero": "Test1"
  },
  {
    "id": 5,
    "fichero": "Test12"
  },
  {
    "id": 11,
    "fichero": "Test8"
  },
  {
    "id": 12,
    "fichero": "Test9"
  },
  {
    "id": 13,
    "fichero": "Test prueba"
  },
  {
    "id": 14,
    "fichero": "WEFWEF"
  }
]
```

Figura 32. *Resultado de la petición GET*

Por otro lado, para eliminar el test con id con valor igual a 14 la petición sería:

```
POST http://localhost:8080/test/deleteTest/14
```

Figura 33. *Petición POST*

5.2 Funciones y procedimientos almacenados

En el apartado 4.1 vimos las ventajas que tenía programar código en la capa de persistencia y por esos motivos es importante saber cómo utilizar ese código desde Spring. Investigando podemos encontrar multitud de formas para acceder a las funciones y procedimientos almacenados de DB2. Aun así, en esta memoria se va a plantear la forma que consideramos más sencilla. Tanto la función como el procedimiento almacenado será el mismo que ya analizamos:

FUNCIONES

El proceso se va a dividir en tres etapas:

1. Creación de una entidad "falsa".

La razón de que la palabra falsa esté entrecomillada es culpa de JpaRepository. Si echamos la vista atrás, recordaremos que JpaRepository necesitaba de una entidad y su clave primaria para poder acceder a los datos. Debido a esto, crearemos una entidad sencilla donde la clave primaria sea un mensaje de error, por ejemplo, y le daremos la opción a Hibernate que decida la estrategia de generación de la clave primaria.

```
@Entity
public class FuncExisteTabla {

    @Id
    @GeneratedValue
    private String msgError;

    public String getMsgError() {
        return msgError;
    }

    public void setMsgError(String msgError) {
        this.msgError = msgError;
    }

}
```

Figura 34. *Entidad falsa para llamar a una función*

2. Consulta SQL desde un repositorio.

Ahora que ya tenemos el parámetro que necesita JpaRepository, procedemos a crear el repositorio que extenderá de JpaRepository. En cuanto a la clave primaria, también será "falsa" debido a que no es una entidad como conocemos, si no un mero trámite para poder realizar la consulta SQL.

```
@Repository("funcExisteTablaRepository")
public interface FuncExisteTablaRepository extends
JpaRepository<FuncExisteTabla, Long> {

    @Query(nativeQuery = true, value =
        "SELECT FUNC_EXISTE_TABLA(:ParamNombreTabla) FROM SYSIBM.SYSDUMMY1")
    int funcExisteTabla(@Param("ParamNombreTabla") String paramNombreTabla);

}
```

Figura 35. *Repositorio con consulta SQL para llamar a una función DB2 desde Java*

El atributo *nativeQuery* indica a JPA que la consulta no contiene nada de Java y permite que la ejecución sea optimizada por el ORM. De esta forma, todas aquellas consultas que sean puras de DB2 y se marquen como verdaderas en ese atributo se ejecutarán con mejor rendimiento.

Fijándonos en los detalles del repositorio, es apreciable que hemos optado por una clave de tipo *Long* como podría haber sido otra cualquiera por lo explicado anteriormente. Por otro lado, en Java seguimos apoyándonos en la tabla virtual que posee DB2 para realizar pruebas llamada SYSDUMMY1.

3. Inyección de dependencias.

La última etapa es poder ejecutar esta función. Como es rutina, lo primero será inyectar la dependencia del repositorio en el servicio que deseemos:

```
@Autowired
@Qualifier("funcExisteTablaRepository")
private FuncExisteTablaRepository funcExisteTablaRepository;
```

Figura 36. *Inyección de dependencias de la función-repositorio*

Y, por último, llamar a la función mediante la siguiente instrucción:

```
this.funcExisteTablaRepository.funcExisteTabla("TABLA_EJEMPLO");
```

Figura 37. *Ejecución de la función desde Java*

PROCEDIMIENTOS ALMACENADOS

Dado que el proceso se asemeja, en gran medida, a la llamada de una función, únicamente destacaremos las anotaciones nuevas que debemos conocer.

En primer lugar crearemos la entidad “falsa” de la misma forma que hemos mencionado. Una vez creada, hay que añadir las anotaciones que nos permiten construir el procedimiento desde Spring:

```
@Entity
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(
        name = "ProcAddColumnaTabla",
        procedureName = "PROC_ADD_COLUMNA_TABLA",
        parameters = {
                @StoredProcedureParameter(mode = ParameterMode.IN, name = "ParamTabla", type = String.class),
                @StoredProcedureParameter(mode = ParameterMode.IN, name = "ParamColumna", type = String.class),
                @StoredProcedureParameter(mode = ParameterMode.IN, name = "ParamTipo", type = String.class),
                @StoredProcedureParameter(mode = ParameterMode.OUT, name = "MsgError", type = String.class)
        })
})
public class ProcAddColumnaTabla { ... }
```

Figura 38. *Anotaciones para la entidad del procedimiento almacenado*

Dentro de la anotación @NamedStoredProcedureQueries insertamos la consulta al procedimiento. Para ello lo más importante es asignar el atributo *procedureName* y completar de forma que coincidan los parámetros. Añadiendo si son de entrada o salida, el nombre de cada uno de ellos y el tipo.

Por último, en el repositorio aparece una anotación nueva muy importante llamada @Transactional. Dado que en el procedimiento almacenado escribimos la palabra reservada ATOMIC, necesitamos que desde Java respete esta decisión también. Es por esto por lo que, gracias a la anotación, nos aseguramos de que la ejecución sea completa, integra y se comporte como un conjunto de instrucciones que no pueden verse alteradas por otras parte del código. Aunque en el procedimiento no hubiésemos marcado ATOMIC, desde Java si podemos usar @Transactional, pero no a la inversa. El resultado final y con el cual cerramos este subapartado quedaría así:

```
@Repository("procAddColumnaTablaRepository")
public interface ProcAddColumnaTablaRepository extends JpaRepository<ProcAddColumnaTabla, Long> {

    @Transactional
    @Procedure(name = "ProcAddColumnaTabla")
    String procAddColumnaTabla(
            @Param("ParamTabla") String paramTabla,
            @Param("ParamColumna") String paramColumna,
            @Param("ParamTipo") String paramTipo
    );

}
```

Figura 39. *Repositorio que implementa un procedimiento almacenado en DB2*

5.3 Implementación de Thymeleaf

Entender el patrón MVC es vital para poder trabajar con Thymeleaf. Gracias a este motor de plantillas, nos introduciremos en la capa de la Vista para presentar nuestra información al usuario. Es importante saber que Thymeleaf solo añade ciertos atributos a las etiquetas del lenguaje HTML para facilitar al programador algunas tareas. Debido a esto, es necesario tener nociones de este lenguaje basado en etiquetas para poder continuar.

Antes de conocer las características del controlador y de la vista, hay que destacar que en la implementación del API Rest vimos el tipo de controlador @RestController. Esta anotación es la equivalencia de @Controller y @ResponseBody. Dado que ahora usaremos la interfaz Model para enviar datos a la vista y no vamos a enviar los datos en formato JSON, que es lo que hace la anotación @ResponseBody, usaremos el tipo de controlador @Controller.

Los métodos que implementemos tendrán que seguir estableciendo una subruta y un tipo de solicitud (GET, POST, …). En lo que se diferenciarán será que devolverán el nombre de la vista y se añadirán atributos a la interfaz Model, que serán los objetos con los que trabajemos.

Para facilitar la comprensión se dejarán dos ejemplos, entre distintas variantes que se permiten, de cómo completar los controladores:

```
@GetMapping(value = "/registrarAlumno")
public String showAlumnoFormulario(Model model) {
    model.addAttribute("alumno", new AlumnoModel());
    return "registrarAlumno";
}
```

Figura 40. *Ejemplo de controlador Thymeleaf 1*

```
@GetMapping(value = "/registrarAlumno")
public ModelAndView showAlumnoFormulario() {
    ModelAndView mav = new ModelAndView();
    mav.addObject("alumno", new AlumnoModel());
    mav.addViewName("registrarAlumno");
    return mav;
}
```

Figura 41. *Ejemplo de controlador Thymeleaf 2*

Tanto en la primera como en la segunda, se instancia el objeto para que pueda ser usado por la vista. En el segundo caso, en vez de usar Model, agrupamos en un solo objeto el modelo y la vista.

Dada la extensión de la documentación de Thymeleaf, se procede a destacar los atributos más comunes y comentar para qué se usan. Previo a eso, veremos que los los atributos de este motor se caracterizan por usar el prefijo “th:” para diferenciarlos del resto.

Atributos más importantes: [18]

th:action=@{/ruta} realiza la acción de enviar los datos del formulario a la ruta que se le indique.

th:object=${objeto} establece el objeto del controlador al cual añadiremos propiedades.

th:field=*{atributo} se le indica el atributo del objeto que queremos relacionar.

th:each="variable: ${lista}" sirve para iterar sobre una lista de objetos.

th:text="${alumno.nombre}" muestra el texto que tenga asignada la propiedad del objeto.

Ahora que ya sabemos qué hacen les daremos un contexto mediante código:

```
<form th:action="@{/registrarAlumno}" th:object="${alumno}" method="post">
  <fieldset>
    <legend>Registrar alumno</legend>
    <table>
      <tr>
        <td><label>Nombre: </label></td>
        <td><input type="text" th:field="*{nombre}" /></td>
      </tr>
      <tr>
        <td><label>Apellidos: </label></td>
        <td><input type="text" th:field="*{apellido}" /></td>
      </tr>
      <tr>
        <td><label>DNI: </label></td>
        <td><input type="number" th:field="*{dni}" /></td>
      </tr>
      <tr>
        <td><label>Edad: </label></td>
        <td><input type="number" th:field="*{edad}" /></td>
      </tr>
      <tr>
        <td>
          <button type="submit">Registrar</button>
        </td>
      </tr>
    </table>
  </fieldset>
</form>
```

Figura 42. *Código HTML usando atributos Thymeleaf*

El resultado de esta vista sería el siguiente:

Registrar alumno

Nombre:

Apellidos:

DNI:

Edad:

Registrar

Figura 43. *Formulario creado con Thymeleaf*

Cierto es que el objetivo principal de la memoria no es centrarse en la capa de la vista y que es tarea del programador conocer sus necesidades. Aunque se haya mostrado unas anotaciones simples para dar a conocer está tecnología, la realidad es que el uso de Thymeleaf se ha incrementado y en el ámbito laboral sí se usa esta alternativa.

La principal ventaja que encontramos en esta elección es que es una forma de unir a programadores frontend y backend. Esto se explica si tenemos en cuenta que las alternativas más populares suelen ser con frameworks que no tienen contacto con Spring dentro del mismo proyecto. El hecho de que esté integrado en Spring permite que ambos tipos de desarrolladores trabajen en un mismo camino dentro del proyecto. En caso de querer mejorar la vista, siempre se puede contar con tecnologías como CSS para darle un estilo más atractivo y JavaScript para añadir lógica más avanzada a la vista.

5.4 API Gateway con Spring

Primeramente, antes de empezar con la explicación, pongámonos en situación. Una vez explicado el objetivo, las desventajas y el uso que se le va a dar, tenemos que imaginarnos que tenemos distintos microservicios en diversos puertos que se comunican entre ellos y ofrecen una respuesta a petición del usuario.

La implementación se fundamenta en crear un proyecto especializado en realizar esta función. Para construir el proyecto seguiremos la misma forma que se explicará en el apartado de Implementación. A raíz de esto, de ahora en adelante, daremos por sentado que ya tenemos el proyecto creado. Ya sea porque sabíamos previamente cómo hacerlo o porque hemos leído ese apartado.

El cambio más importante que debemos hacer es cambiar el archivo, que se encuentra en este proyecto, cuyo nombre es el de "application.properties" por "application.yml". Este formato de archivo se utiliza para estructurar el contenido de forma jerárquica. La validez es la misma que el anterior pero, teniendo en cuenta que este tipo de proyectos pueden llegar a contener una configuración muy amplia, hemos considerado favorecer un formato de lectura más rápido.

Todo microservicio debe tener un puerto asignado y diferente al resto. El puerto que utiliza Spring por defecto es el 8080. Suponiendo que tenemos cuatro microservicios ya implementados cuyos puertos son el 8001, 8002, 8003 y el 8004, estableceremos el puerto del gateway en el 8080. De esta forma, estaremos obligados a cambiar el puerto por defecto de los posibles nuevos microservicios ya que éste estará en uso. Esta práctica se menciona aquí, pero queda a elección del programador qué estrategia seguir.

En el ejemplo de configuración, veremos como el último microservicio parece que solo acepte las peticiones que empiecen por el prefijo "/public". La realidad es que realmente funciona igual que el resto y acepta todas las peticiones, ya que utiliza un filtro para ignorar el primer prefijo. La variedad de filtros también permitiría desarrollar otra memoria debido a la cantidad de alternativas que existen.

```yaml
server:
  port: 8080

spring:
  application:
    name: gateway
  cloud:
    gateway:
      routes:
        - id: microservicio1
          uri: http://localhost:8001/
          predicates:
            - Path=/**

        - id: microservicio2
          uri: http://localhost:8002/
          predicates:
            - Path=/**

        - id: microservicio3
          uri: http://localhost:8003/
          predicates:
            - Path=/**

        - id: microservicio4
          uri: http://localhost:8004/
          predicates:
            - Path=/public/**
          filters:
            - StripPrefix=1
```

Figura 44. *Archivo de configuración application.yml del API Gateway*

Utilizaremos *server.port* para establecer el número del puerto de nuestro proyecto. Dentro del apartado *routes* añadiremos tantas como microservicios deseemos unificar. El id es un identificador para nosotros los programadores y en la *uri* escribiremos la ruta destacando la máquina, local o una ip remota, acompañada del puerto.

Si anteriormente enviábamos la petición *http://localhost:8003/test* y nos devolvía cierto resultado, ahora podremos obtener ese mismo resultado ejecutando *http://localhost:8080/test*, y así con el resto de los microservicios. De esta forma, queda unificado en un mismo puerto y en un mismo proyecto todas las redirecciones.

La configuración presentada es la más sencilla. La variedad de opciones que permite esta implementación es increíblemente extensa a la par que permite mucha concreción. Se deja en manos del programador conocer sus requisitos e investigar cómo adaptar esta solución a su situación.

5.5 Eureka Server con Spring

La implementación de este servicio se recomienda integrarla en algún microservicio que siempre se encuentre activo o en uno creado específicamente para realizar esta función. Esto es debido a que el objetivo que perseguimos está relacionado con conocer en todo momento qué microservicios están activos. Por ende, si lo establecemos en un microservicio que suela dejar de funcionar o se reinicie constantemente dejaríamos de conocer esa información.

Si partimos de que ya tenemos microservicios implementados y queremos registrarlos, tendremos que añadir la dependencia Eureka Client en el archivo *build.gradle* de cada microservicio. La dependencias es la siguiente: [19]

```
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client:3.0.3'
```

Figura 45. *Dependencia Eureka Client*

Una vez añadida, debemos refrescar ese archivo de dependencias. Generalmente suele aparecer un icono emergente para realizar dicha acción. Por último, en cuanto al cliente se refiere, tendremos que añadir la anotación @EnableEurekaClient en la clase principal de nuestros microservicios.

Para esta solución, implementaremos el Eureka Server en un microservicio aparte. Podría incluirse en el gateway, pero de esta forma quedan ambos proyectos más limpios y reutilizables. La tarea principal es añadir la dependencia de Eureka Server al igual que hicimos con los clientes. Además, en nuestro archivo *application.yml* añadiremos que no se registre así mismo y una zona por defecto que se requiere para su funcionamiento. Suele estar formada por la dirección de la máquina, acompañada del puerto y la palabra "eureka": [20] [21]

```
server:
  port: 8761

eureka:
  client:
    registerWithEureka: false
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

Figura 46. *Configuración básica del Eureka Server*

Esta serie de configuraciones son las mínimas requeridas. Si lanzamos todos nuestros microservicios, podremos acceder al *dashboard* donde se registra toda la información que puede proporcionarnos este microservicio. Para acceder a él únicamente debemos escribir la *defaultZone* en nuestro navegador. En las siguientes imágenes analizaremos los metadatos que recoge:

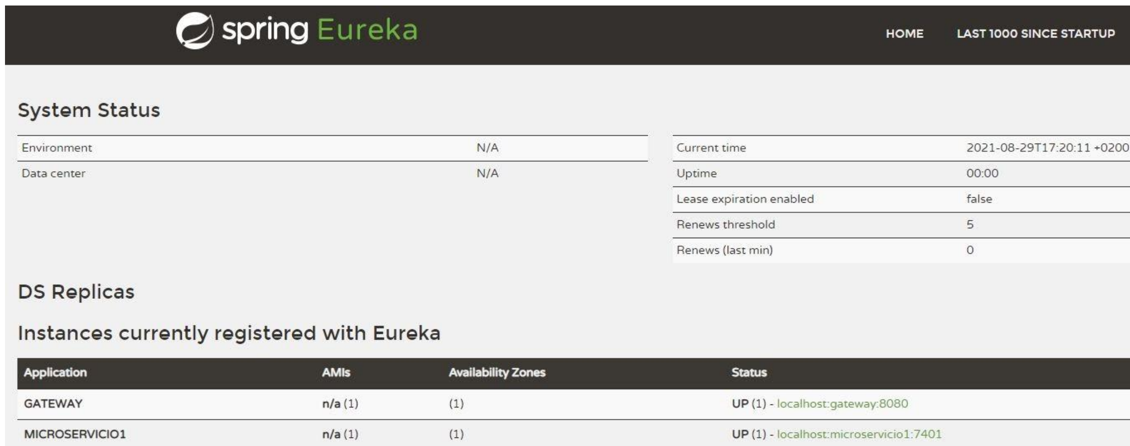

Figura 47. *Dashboard Spring Eureka 1*

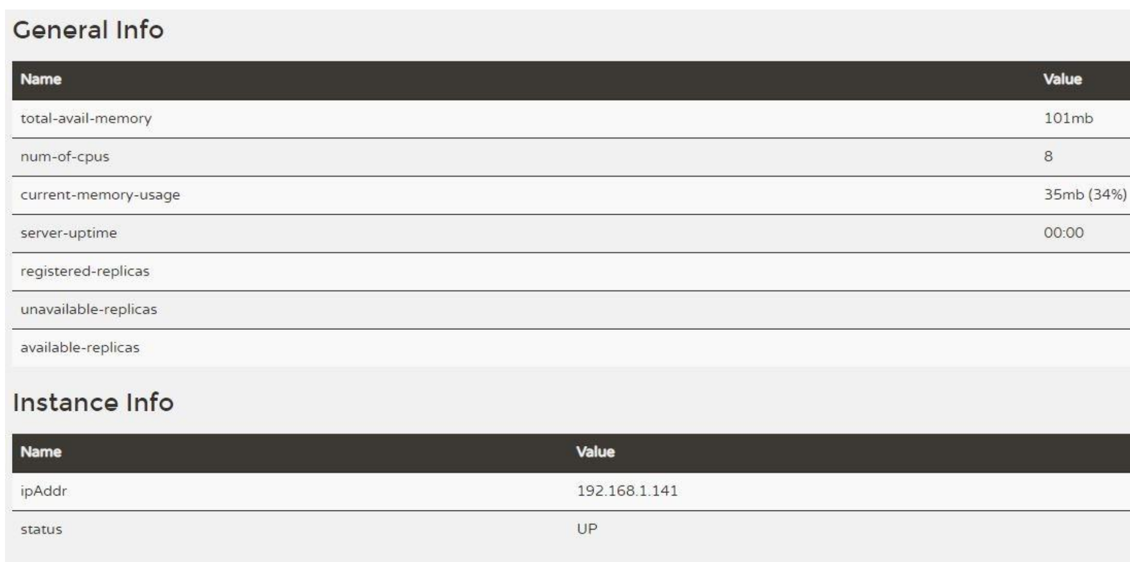

Figura 48. *Dashboard Spring Eureka 2*

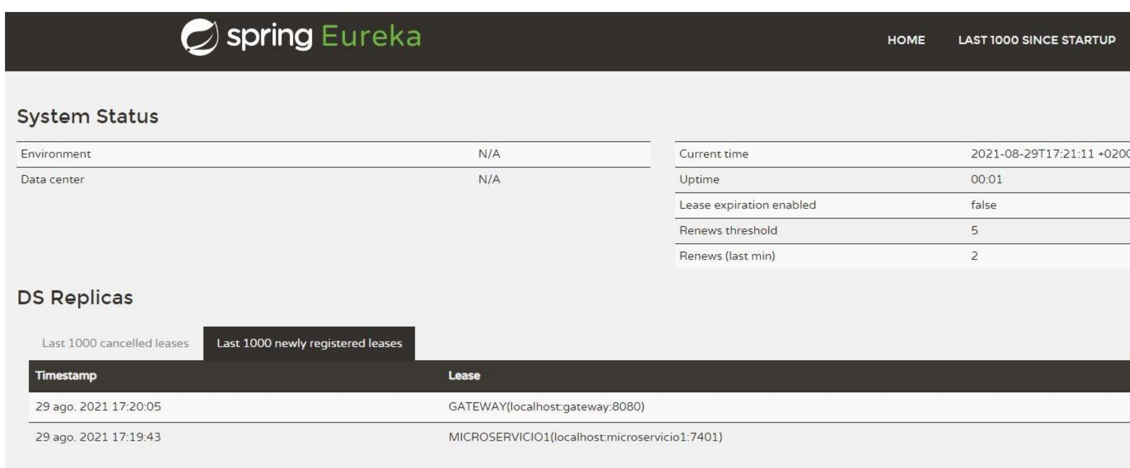

Figura 49. *Dashboard Spring Eureka 3*

La información que recoge que consideramos más importante es la siguiente:

- La localización de los microservicios y si su estado está definido como activo o no.
- La cantidad de memoria disponible.
- Número de *cpus* que están en ejecución.
- Porcentaje de memoria que está en uso.
- El tiempo que se encuentra encendido el microservicio.
- La dirección IP de la máquina en la que se está ejecutando y se encuentra activo.
- El registro de los últimos microservicios registrados. Esto nos proporciona información de si se ha registrado alguno, después ha fallado y se ha vuelto a registrar. Es decir, conocemos si se caen microservicios.

Aunque pueda parecernos poca información, el simple hecho de conocer si un microservicio se encuentra activo y su ubicación nos facilita muchas tareas en entornos complejos. Obviamente sería ideal que se pudieran levantar los microservicios, desde el *dashboard*, en caso de que fallasen, y otras muchas tareas que se nos puedan ocurrir en función de las necesidades de nuestra situación.

Otro objetivo de esta tarea es mostrar la facilidad con la cual Spring nos proporciona un abanico de herramientas con las que complementar nuestro backend. Cierto es que, esta última tarea, no es la más necesaria ni un requisito indispensable para los programadores backend. Pero, por otro lado, queríamos complementar esta memoria, que abarca los problemas principales que todo programador backend debe ser capaz de implementar, con una tarea que fomente crear proyectos en base a nuestra creatividad.

Analizadas las implementaciones de nuestras tareas, podemos afirmar que la más importante y extensa es el servicio REST. Para ello, necesitaremos conocer el patrón MVC y la característica más importante de Spring como lo es la inyección de dependencias. Si se complementa con poder inyectar la funcionalidad integrada en la capa de persistencia, lograremos programar cualquier tipo de requisito que se nos pida.

6. Implementación

Una vez vistas las tareas principales de esta memoria, procedemos a explicar detalladamente los pasos a seguir para poder construir y desarrollar nuestros proyectos. Principalmente veremos cómo instalar las tecnologías mencionadas y la configuración básica que debemos conocer.

6.1 Instalación de DB2

En primer lugar, instalaremos la base de datos IBM DB2. Para ello, IBM nos obliga a registrarnos en la página web oficial. Después de ello, accederemos al catálogo de productos que nos ofrece. Buscamos IBM DB2 Database y la seleccionamos. Descargamos el instalador que sea compatible con nuestra máquina.

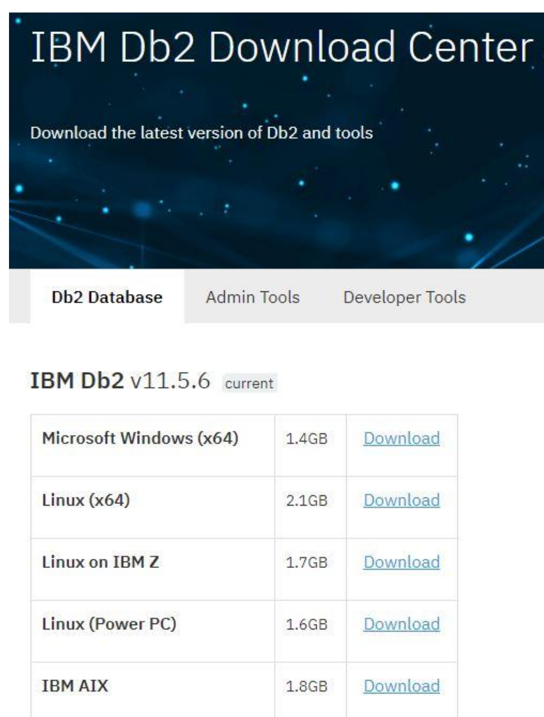

Figura 50. *Instalación DB2 – Descargar el instalador*

Una vez descargado el archivo comprimido, lo descomprimimos en el directorio que queramos. Al final de las carpetas deberemos tener un archivo llamado "setup". Para iniciar la instalación ejecutaremos ese archivo en modo administrador. Dado que es la primera vez que realizamos este proceso, seleccionaremos en el apartado de la izquierda "Instalar un producto" y continuaremos aceptando los siguientes pasos marcando las opciones por defecto. En usuario y contraseña dejaremos el usuario db2admin aunque esto queda a elección del programador.

Si todo ha sido correcto, en los iconos ocultos de la barra de tareas deberíamos tener la instancia DB2 con el nombre de la copia DB2COPY1 de la siguiente forma:

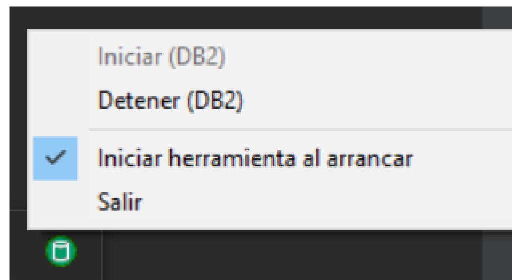

Figura 51. *Instancia DB2 en los iconos ocultos de la barra de tareas*

Si partimos de la base que ya tenemos instalado SqlDbx o DBeaver podríamos acceder a la base de datos por defecto que nos ofrece DB2 llamada SAMPLE.

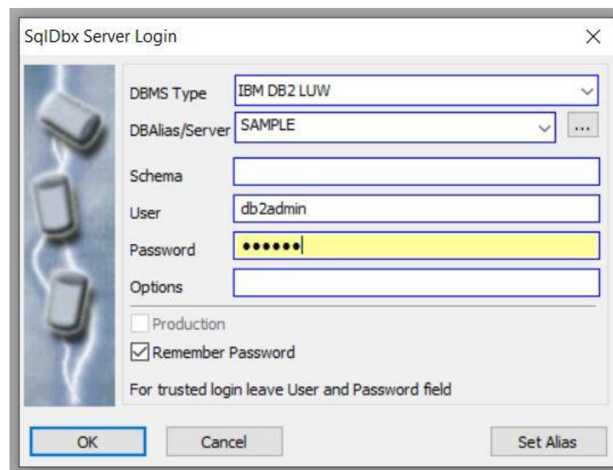

Figura 52. *Consulta de la base de datos SAMPLE con SqlDbx*

6.2 Creación de microservicios

Para no hacer más extensa la memoria de lo necesario, daremos por sabido que el programador conocerá la forma de instalar y configurar Java, IntelliJ, SqlDbx, DBeaver, IBM Data Studio, Git y Fork según sus necesidades. Partiendo de esta base, solo faltaría conocer cómo crear los proyectos Spring.

Una de las filosofías más importantes de esta guía es facilitar el aprendizaje para implementar nuestras ideas. Si además de esto, contamos con tecnologías que nos ahorran el gastar tiempo configurando los proyectos entonces se convierte en una marco de trabajo idóneo. Por este motivo, usaremos Spring Initializr [5]. Como bien se explicó en el apartado de Estado del arte, esta utilidad web nos permite crear el esqueleto de nuestro proyecto Spring Boot de la forma más simple, ahorrándonos especialmente tiempo en la gestión de la configuración.

Una vez configurado el proyecto, haciendo uso de la explicación que ahora veremos, se generará una archivo comprimido que tendremos que descomprimir e importar en nuestro IDE. En nuestro caso IntelliJ.

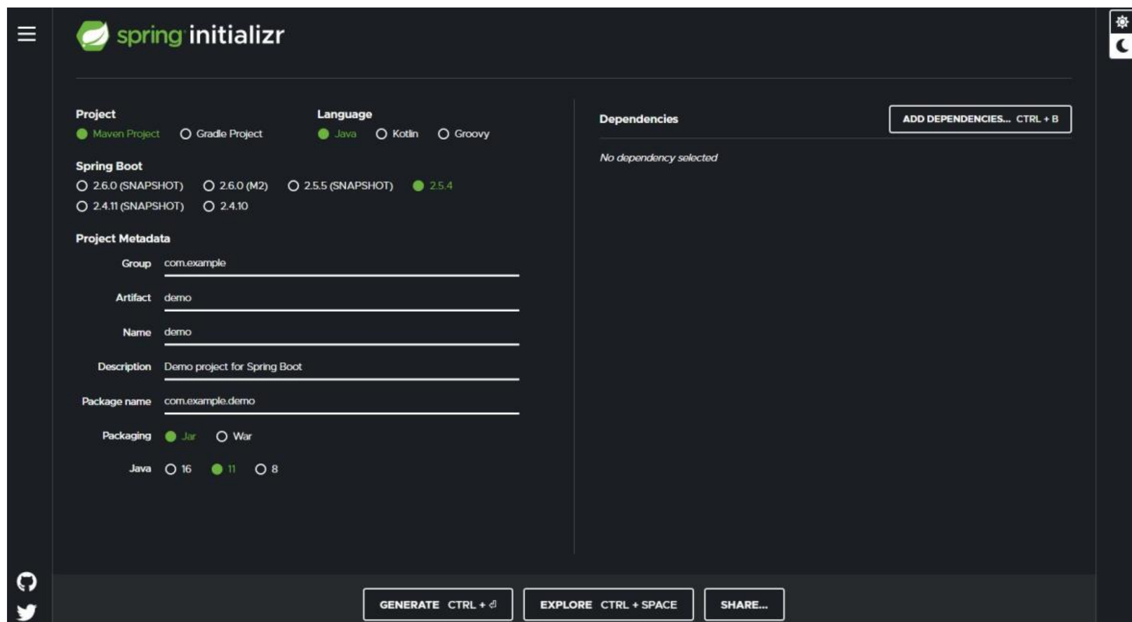

Figura 53. *Interfaz de Spring Initializr*

Teniendo en cuenta que si configuramos este paso como es debido nos podremos poner a escribir código directamente, se procede a detallar la explicación.

Project

En este apartado nos da a elegir entre Maven y Gradle. Esta dicotomía ha sido fruto de debate durante un largo tiempo. Ambas herramientas gestionan la construcción del proyecto y sus dependencias. Aunque es sabido que Maven ha sido la elección favorita durante muchos años en proyectos Java, optaremos por Gradle. La explicación es la siguiente:

- Elimina el archivo de configuración pom.xml. Es decir, una configuración menos que tratar.
- Gestiona el ciclo de vida del proceso software de forma más completa. Desde la compilación a la implementación.
- La configuración de Maven es más tediosa y tendida en el tiempo.
- Permite crear scripts de forma más sencilla. Se verán ejemplos de scripts Gradle en el apartado de Trabajos futuros.
- Con Maven, se imprimen más mensajes por la terminal cuando se ejecuta el proyecto.
- El lanzamiento del proyecto con Gradle es más rápido.

Spring Boot

Seleccionamos la versión 2.5.4 ya que las otras son más antiguas o están en desarrollo.

Project Metadata

- Group: representa a la organización autora del proyecto.
- Artifact: suele recibir el nombre que por el que se conoce el proyecto en sí.
- Package name: se genera automáticamente con group y artifact.
- Packaging: war se utiliza para aplicaciones web. En el ámbito laboral, para crear microservicios se opta por jar.
  La forma de trabajar con jar en la realidad suele ser: ejecutar la tarea Gradle "jar", mover el archivo con el código compilado a la máquina remota con la que trabajemos y disponer ahí de los nuevos cambios para que los frontend apunten a ese jar.
- Java: la versión 8 dejó de tener soporte en enero de 2019. En caso de que no se trabaje con clientes cuya versión siga siendo la 8, se recomienda utilizar la 11. Los cambios tan notorios que ha habido entre versiones no se relacionan con el objetivo de esta memoria.

En el siguiente subapartado cerraremos la configuración de Spring Initializr abordando el asunto de las dependencias.

6.3 Dependencias

Dependiendo del tipo de proyecto, seleccionaremos unas dependencias u otras. En vista de que a lo largo de esta memoria hemos abordado 4 tipos, los analizaremos individualmente para destacar qué dependencias se necesitarían.

Servicio REST

- Spring Web. Cubre las necesidades para construir el servicio REST, usar el patrón MVC y, por si fuera poco, trae consigo un servidor Apache Tomcat embebido.
- Spring Data JPA. Persiste los datos utilizando el conjunto de reglas de su API conjuntamente a Hibernate.
- IBM DB2 Driver. Concede acceso a la base de datos de IBM DB2.

API Gateway

- Gateway. Proporciona una forma efectiva de enrutar las API, además de poder administrar la seguridad, el monitorio y conocer ciertas métricas. Esta dependencia no puede añadirse en un mismo proyecto donde ya esté la dependencia Spring Web. Por este motivo, entre otros, el proyecto que abarque la gateway estará aparte del resto de microservicios.

Eureka Server

- Eureka Server. Únicamente se añadirá una vez y su localización estará en el microservicio que queramos que registre el resto. Puede estar en uno a parte, en el gateway, en uno ya implementado…
- Eureka Discovery Client. Esta dependencia se añadirá a los microservicios que deseemos poder registrar y que sean localizados para el servidor.

Thymeleaf

- Su dependencia se añadirá en el proyecto que contenga el servicio REST que quiera montar la capa de la vista con este motor de plantillas. En el resto de los proyectos de esta memoria no tendría utilidad.

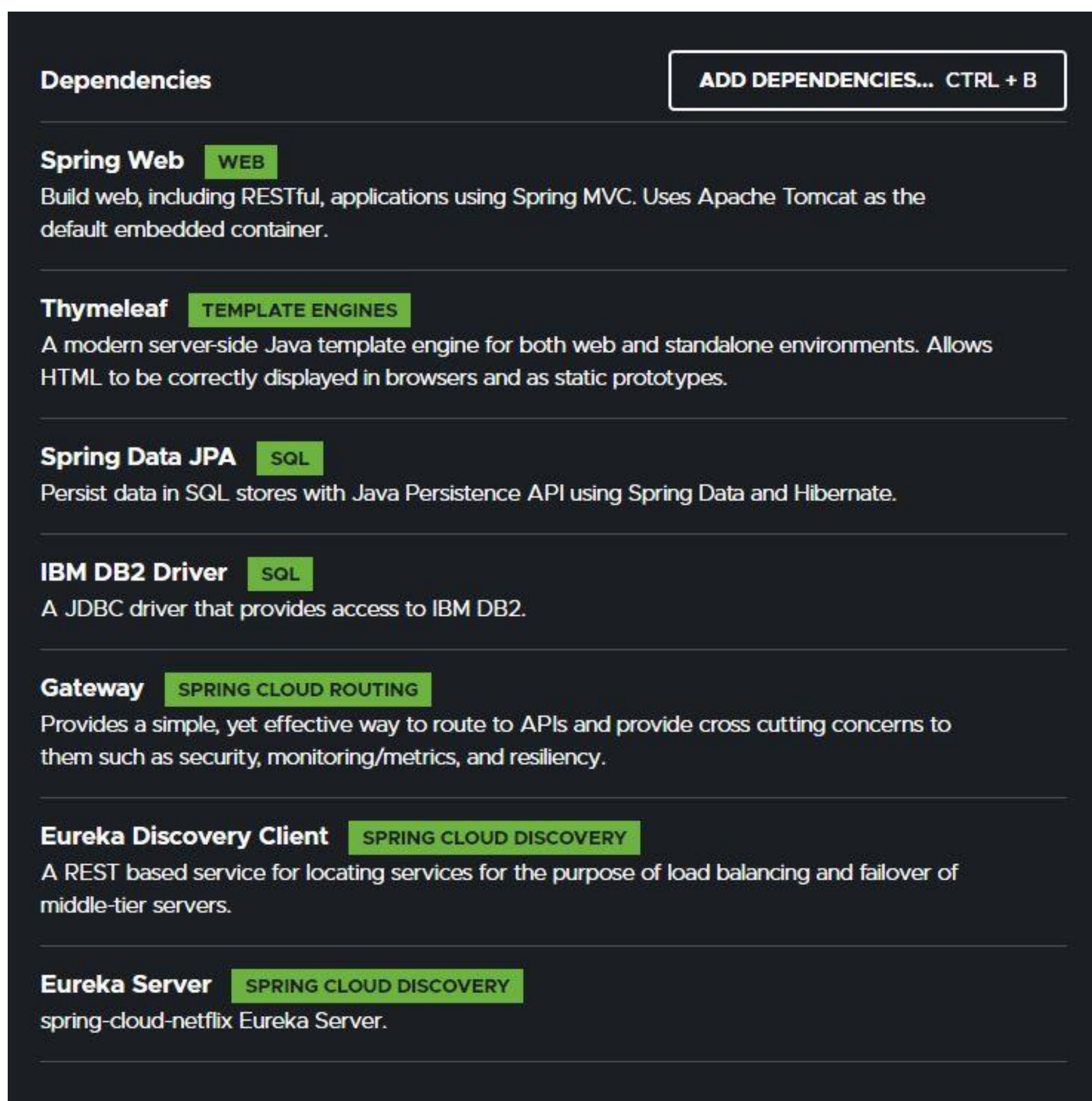

Figura 54. *Lista de dependencias mencionadas*

Aunque por extensión de la memoria no se ha hablado de ella, Spring ofrece una dependencia llamada Spring Security que nos ofrece soporte completo en la autentificación como la autorización. También protege contra ataques de sesiones, falsificación de solicitudes y se integra perfectamente con Spring Web MVC.

Bajo mi punto de vista, en el ámbito laboral, si empiezas a trabajar en una empresa seguramente ya dispongan de un sistema de seguridad implementado. Esto suele ser de las primeras tareas y existen multitud de herramientas para hacerlo. En caso de que no sea así, siempre puede contar con Spring Security y proteger su aplicación de una forma completamente fiable.

6.4 Configuración inicial

La estructura común que poseen nuestros microservicios es la siguiente:

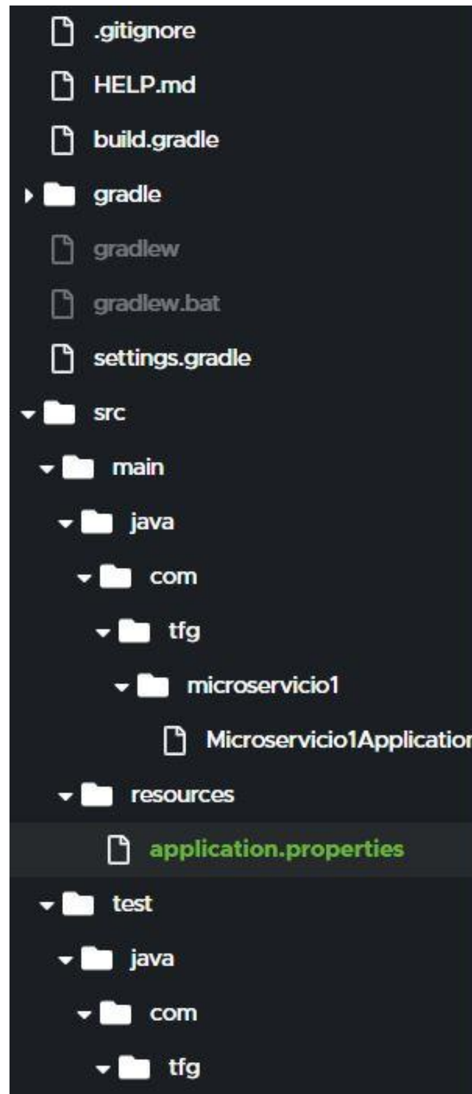

Figura 55. *Estructura base de los microservicios*

En este subapartado, estudiaremos las propiedades clásicas más importantes y que necesitaremos si queremos satisfacer las tareas descritas en la memoria. Para ello, en el archivo *application.properties* escribiremos:

- server.port=X

Se debe colocar el número del puerto con el cual la aplicación de conectará al servidor.

- spring.datasource.url=jdbc:db2://localhost:50000/SAMPLE:currentSchema=DB2ADMIN.

En esta propiedad se configura la conexión con la base de datos. Si vamos de izquierda a derecha, se observa que establecemos el tipo de base de datos (DB2), la máquina donde se encuentra ubicada la base de datos (localhost o remota), el puerto de la base de datos que no hay que confundir con el puerto del microservicio (por defecto, el 50000). Por último, el nombre de la base de datos (SAMPLE) y el esquema donde se encuentra dicha base de datos (DB2ADMIN).

- spring.datasource.username=db2admin

En la instalación de DB2 ya introducimos el usuario con el cual nos identificaremos. Para acceder a ella estableceremos el mismo.

- spring.datasource.password=prueba

En caso de querer proteger el acceso y haber configurado una contraseña, introduciremos la que creamos más conveniente.

- spring.datasource.driver-class-name=com.ibm.db2.jcc.DB2Driver

Aunque en ocasiones Spring puede deducir el driver que se requiere desde la propiedad *spring.datasource.url*, es recomendable introducirlo para facilitar la tarea y evitar errores.

- spring.jpa.hibernate.ddl-auto=X

Esta propiedad acepta cuatro tipos de valores: create, create-drop, validate, and update. Las dos primeras se suelen utilizar si estamos realizando pruebas. Cada vez que se ejecuta la aplicación, Spring automáticamente generará las entidades que tengamos anotadas como @Entity, sus relaciones, etc. Si a lo largo de la ejecución guardamos registros estos se eliminarán cuando se cancele el proceso. El valor "*validate*" se utiliza en la fase de pruebas o de calidad para verificar que los scripts de la base de datos son correctos. Por último, "*update*" actualiza la base de datos añadiendo columnas, restricciones, etc. Pero nunca elimina nada que se haya introducido en ejecuciones anteriores.

Cabe destacar, que el valor por defecto de esta propiedad es "*none*" y en la realidad, en entornos empresariales, esta propiedad no se suele especificar debido a que no es necesaria para acceder a los datos. Se entiende muy bien si nos ponemos en la situación de que una empresa ya posee una base de datos y desea implementar su lógica con Spring. Si se estableciese esta propiedad, podría eliminar toda la información que contiene.

- spring.application.name=X

Su valor registra el nombre con el cual identificaremos la aplicación.

- spring.jpa.show-sql=false | true

Su valor por defecto es falso. En caso de añadir esta propiedad con valor igual a cierto, cuando realicemos una consulta u operación en la base de datos mediante lógica del programa también aparecerá dicha consulta en la terminal. Esta propiedad viene muy bien para identificar en qué falla alguna consulta u operación.

7. Pruebas

Una de las tareas más importantes de todo programador backend es comprobar que el código que implementa funciona. Especialmente para tener la seguridad de que lo que ha hecho da el resultado esperado, pero también para asegurarnos de que en el futuro siga funcionando. Es decir, si otro programador dentro de cierto tiempo modifica la lógica que añadimos y comprobamos en el pasado, los test que creamos en su momento deberán devolver el mismo resultado que obtuvimos.

En este apartado, nos centraremos en añadir validaciones a nuestras entidades antes de realizar cualquier operación CRUD en la base de datos. Estas validaciones pueden nacer como consecuencia de dos fenómenos. En primer lugar, querer añadir más restricciones a nuestras entidades. En segundo lugar, disponer de una base de datos creada hace años con un gran número de tablas, en la cual no se establecieron relaciones entre entidades por desconocimiento o por añadir simplicidad e independencia a las tablas. Por este motivo, entre otros muchos ejemplos, gracias a las validaciones se pueden añadir esas comprobaciones para saber si existe ese valor en otra entidad.

7.1 Esquema de las validaciones

Debido a que se van a añadir varias clases con sus respectivas relaciones, para explicar cómo realizar las validaciones correctamente, se deja adjunta una imagen y en función de ella procederemos a explicar los componentes. En el siguiente subapartado se detallará su implementación:

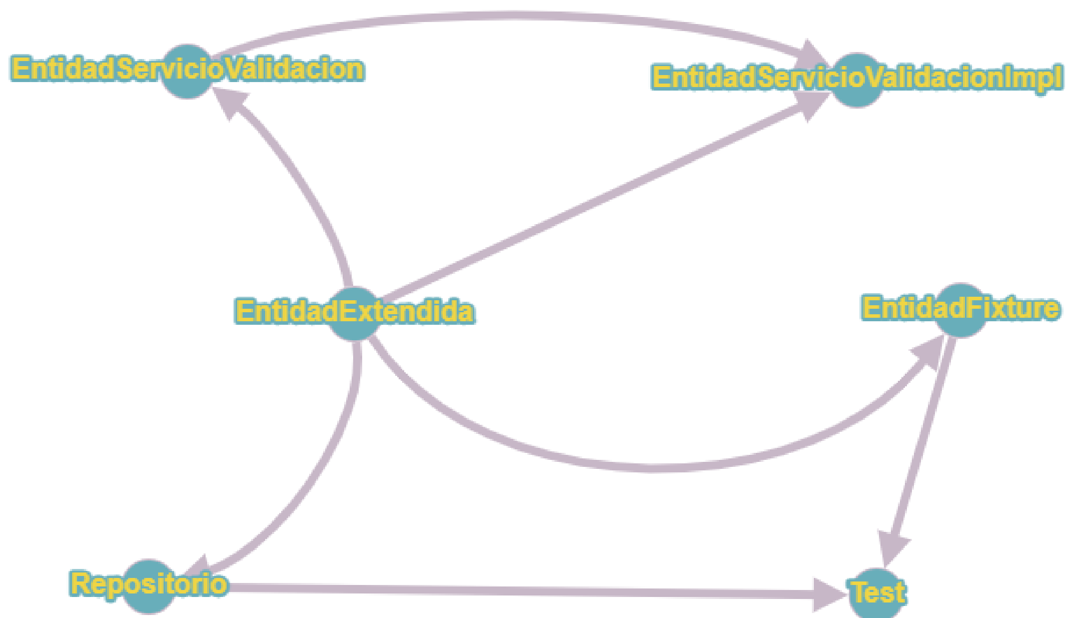

Figura 56. *Esquema de la estructura de las validaciones*

Repositorio

Este componente del esquema es el mismo que ya hemos explicado, únicamente se añadirá una nueva funcionalidad para que contenga métodos que devuelvan datos nuevos con los que realizar las pruebas. Es decir, si queremos comprobar las validaciones que se lanzan cuando se inserta un registro, necesitaremos generar objetos cuya clave primaria no se repita cada vez que se ejecutan los test. En caso de que por ejemplo sea un integer la clave primaria, este repositorio contendrá un método que accederá a la base de datos para consultar cual es el valor máximo de la clave primaria y retornará ese valor más uno.

Entidad extendida

Por entidad extendida entendemos la entidad que extiende la super clase ya mencionada en esta memoria. A esta entidad extendida se le añadirá la dependencia del servicio que contendrá las validaciones que crearemos. Además, gracias a las anotaciones @PrePersist, @PreRemove, @PreUpdate, @PostPersist, @PostRemove, @PostUpdate podremos especificar en qué momento se comprueba cada validación del servicio de validaciones.

EntidadServicioValidacion

Esta interfaz contendrá los métodos de las validaciones.

EntidadServicioValidacionImpl

Es el servicio principal. En él estará el código de la lógica de las validaciones. Se realiza en una clase aparte y no en la clase extendida como se podría pensar al estar accesibles en ellas las anotaciones @Pre y @Post para una mejor legibilidad del código. Asimismo, el número de validaciones que se pueden llevar a cabo en la realidad pueden ser cuantiosas, es por ello por lo que se toma esta decisión de independizar las validaciones y añadir el menor número de líneas de código en las clases extendidas.

EntidadFixture

Se denomina fixtures a los datos de prueba con los que trabajaremos en los test. Se distingue entre los datos que pueden venir del frontend y los que, el propio equipo de backend, genera para realizar pruebas en sus test.

Realmente esta clase satisfará el estereotipo de servicio que contendrá la lógica de los datos de prueba. Por ejemplo, si cada vez que ejecutamos los test necesitamos crear un nuevo artículo al azar con el que jugar cuya clave primaria sea el nombre del artículo, esta clase generará ese nombre del artículo de manera completamente aleatoria.

Test

En esta clase comprobaremos que las validaciones devuelven lo esperado.

En el siguiente subapartado veremos todos estos componentes con mayor profundidad.

7.2 Pruebas unitarias

El caso de prueba con el que analizaremos nuestra primera validación será el siguiente. En primer lugar, existirá una entidad que contenga los distintos tipos de artículos que existen. Por ejemplo, si es una materia prima, una pieza de fabricación, un semielaborado, etc. En realidad los datos que pueda contener la base de datos nos deben ser irrelevantes, ya que la validación será independiente a ellos. A continuación, tendremos otra entidad cuyo contenido almacenará todos los artículos de nuestra aplicación. Recapitulando, dos entidades. La primera TIPOS_ARTICULO y la segunda ARTICULOS.

Para un posterior uso, crearemos el método del repositorio que nos devuelve un nuevo valor para la clave primaria. Éste consulta cuál es el valor máximo y le añade una unidad.

```
@Query(value = "select coalesce(max(t.id),0)+1 from TiposArticuloExtends t")
int getSiguienteIdTiposArticulo();
```

Figura 57. *Consulta que genera una nueva clave primaria*

Se debe conocer que la función COALESCE retorna el primer valor que no sea NULL. Solucionado esto, esto ya tendríamos nuestro repositorio preparado para generar datos de prueba para los test.

El siguiente paso será crear la interfaz del servicio de validaciones. Para ello, crearemos los métodos que más tarde sobrescribiremos.

```
public interface BaseValidaciones<T> {

    void preInsert(final T nuevaEntidad) throws Exception;

    void preUpdate(final T entidadModificada) throws Exception;

    void preDelete(final T entidadBorrada) throws Exception;

    void postInsert(final T nuevaEntidad) throws Exception;

    void postUpdate(final T entidadModificada) throws Exception;

    void postDelete(final T entidadBorrada) throws Exception;

}
```

Figura 58. *Métodos de las validaciones*

De esta forma, la cabecera de nuestra interfaz quedará de la siguiente forma:

```
public interface TiposArticuloEntidadServicioValidacion<T> extends
BaseValidaciones<T> {
}
```

Figura 59. *Interfaz servicio validaciones para los test*

Apoyándonos de la anotación @Service crearemos nuestro servicio de validaciones que extenderá la interfaz recién comentada, obligándonos a añadir los métodos a sobrescribir, y sustituiremos la entidad como parámetro genérico.

Si por ejemplo, una de las validaciones que deseamos implementar es que antes de borrar un registro de la tabla TIPO_ARTICULO compruebe si hay algún ARTICULO que usa ese tipo quedaría tal que así:

```
@Override
public void preDelete(TiposArticuloExtends entidadBorrada) throws Exception {
    this.compruebaSiTipoExisteEnArticulos(entidadBorrada.getId());
}
```

Figura 60. *Ejemplo de nuestra primera validación*

Aunque la implementación del método no se describirá, lo que debe realizar es buscar mediante el repositorio de la tabla de ARTICULOS si existe algún artículo que contenga el identificador de ese tipo. En caso de que exista, devolverá un mensaje de error personalizado lanzando la excepción.

De nada serviría este proceso si no se inyecta esta validación en la clase extendida de TIPOS_ARTICULO.

```
@Transient
@JsonIgnore
@Autowired
@Qualifier("tiposArticuloEntidadServicioValidacionImpl")
private TiposArticuloEntidadServicioValidacion servicioValidaciones;
```

Figura 61. *Inyección de dependencia del servicio de validaciones*

Anotamos este servicio con @Transient para indicar a JPA que no debe persistir este atributo. Al igual que usamos la anotación @JsonIgnore para que no la devuelva al frontend este atributo cuando retornemos un objeto de este tipo.

Ahora solo nos faltaría añadir la anotación que vincula todo para que se valide nuestro requisito.

```
@PreRemove
public void preRemove() throws Exception {
    this.servicioValidaciones.preDelete(this);
}
```

Figura 62. *Validación inyectada en la entidad extendida*

Una vez implementada la validación, el siguiente paso sería comprobar que se ejecuta correctamente y nos devuelve el mensaje de error en caso de que exista algún artículo con ese tipo. Para ello necesitaremos un componente que nos ahorrará mucho código en los test:

TiposArticuloFixture

Es el componente mencionado anteriormente como EntidadFixture. La función de este servicio será generar datos de prueba. Usando la anotación @Service e inyectando la dependencia del repositorio crearemos un método que nos devuelva objetos para los test.

```
public TiposArticuloExtends getNuevoTipo() {
    TiposArticuloExtends tipo;
    try {
        tipo = new TiposArticuloExtends();
        tipo.setId(tiposArticuloRepository.getSiguienteIdTiposArticulo());
        return tipo;
    } finally {
        tipo = null;
    }
}
```

Figura 63. *Ejemplo de fixture*

Test

La clase de test se genera haciendo clic derecho en la clase del servicio de validaciones de la entidad que deseemos y buscando Generate + Test. En él, inyectaremos las dependencias de los servicios fixtures que necesitemos junto a los repositorios de las entidades que deseemos comprobar sus validaciones.

```
@Test
public void borrarTipoArticulo_SiIdExisteEnArticulosNoSePuede() {
    String mensajeEsperado = "Mensaje de error personalizado";
    TiposArticuloExtends tipo;
    MaestroDeArticulosExtends nuevoArticulo;
    try {
        tipo = this.tiposArticuloFixture.getNuevoTipo();
        this.tiposArticuloRepository.save(tipo);
        nuevoArticulo = this.articulosFixture.getNuevoArticulo();
        nuevoArticulo.setTipo(tipo.getId());
        this.articulosRepository.save(nuevoArticulo);

        this.tiposArticuloRepository.delete(tipo);
        AssertionErrors.assertEquals("No debe llegar aquí. Debe entrar a la
excepción", false, true);
    } catch (Exception ex) {
        AssertionErrors.assertEquals("Test correcto", mensajeEsperado,
ex.getMessage());
    } finally {
        tipo = null;
        nuevoArticulo = null;
    }
}
```

Figura 64. *Ejemplo de test que comprueba nuestra validación*

Como es apreciable, primero se crea el tipo a través del fixture que genera datos de prueba. A continuación y se crea el artículo al cual se le asigna el tipo que acabamos de crear. Guardados los dos objetos, se procede a eliminar el tipo y dado que hay un artículo con ese tipo, la validación lanzaría un mensaje de error que debe coincidir con el esperado.

7.3 Desactivar validaciones externas

El objetivo de este apartado es darnos cuenta de un problema que conllevan las validaciones y abordar una solución. Si la mayoría de nuestras entidades poseen validaciones y las entidades están relacionadas entre ellas como es el caso de TIPOS_ARTICULO y ARTICULOS, cuando se realicen test los datos de prueba deben tener en cuenta todas las validaciones previas. Dicho en otras palabras teniendo en cuenta un ejemplo, si en el caso anterior la entidad ARTICULOS también tuviese validaciones, cuando hubiésemos creado el artículo para asignar el tipo y comprobar la validación que estábamos analizando, también tendríamos que haber tenido en cuenta esa validación de la entidad ARTICULOS y el test dejaría de ser específico para la entidad TIPOS_ARTICULO.

Esto supone un problema si se lleva al extremo como pudiese ser el caso en una empresa donde su base de datos contenga decenas o centenares de entidades con validaciones. Al final, para cada validación, todos los datos de prueba tendrían que cumplir con todas las validaciones previas y el programador tendría que conocerlas para esquivarlas. Esto supone un gran retraso en el tiempo para generar casos de test.

Por este motivo, ahora que se conoce el problema, se va a plantear una solución que desactiva las validaciones externas y únicamente tiene en cuenta las validaciones de la entidad que estemos testeando en ese momento.

Para ello, tenemos que conocer la utilidad de una interfaz y una clase que usaremos. La interfaz *BeanDefinitionRegistry*. Esta interfaz es capaz de acceder a los *beans* que se encuentran en el contenedor de Spring que mencionamos cuando explicamos por primera vez la inyección de dependencias. Los *beans* en este caso son los objetos que maneja el contenedor de Spring. Por otro lado, la clase *GenericBeanDefinition* nos proporciona acceso para registrar beans.

La forma en la que utilizaremos lo que acabamos de mencionar seguirá el siguiente proceso. Accediendo al registro de todos los beans, si se encuentra activo nuestro servicio de validaciones lo dejaremos intacto, pero el resto de servicio de validaciones de otras entidades lo eliminaremos y registraremos un servicio de validaciones que no contenga nada en los métodos *prePersist, preRemove, preUpdate, postPersist, postRemove* y *postUpdate* ilustrados en la imagen "Ejemplo de nuestra primera validación" de esta memoria. De esta forma, las únicas validaciones que se tendrán en cuenta serán las del servicio de validaciones de la entidad que estemos ejecutando los test.

La implementación de esta solución tiene como primer paso crear una clase que extenderán todos nuestros test donde se encontrará el método que realice dicho proceso:

```
protected void deshabilitarValidacionesExcepto(String nombreValidacion) {
    BeanDefinitionRegistry registry = (BeanDefinitionRegistry)
AutowireHelper.getApplicationContext().getAutowireCapableBeanFactory();
    GenericBeanDefinition gbd = new GenericBeanDefinition();

    if (registry.isBeanNameInUse("articuloEntidadServicioValidacionImpl")) {
        registry.removeBeanDefinition("articuloEntidadServicioValidacionImpl");
        if (nombreValidacion.contains("articuloEntidadServicioValidacionImpl")) {
            gbd.setBeanClass(ArticulosEntidadServicioValidacion.class);
        } else {
            gbd.setBeanClass(ArticulosEntidadServicioValidacionFake.class);
        }
        registry.registerBeanDefinition("articuloEntidadServicioValidacionImpl", gbd);
    }
    …
}
```

Figura 65. *Desactivación del resto de servicios de validaciones*

Si observamos el código, se puede apreciar que en caso de que estemos validando la entidad TIPOS_ARTICULO la ejecución entrará por el *else* y registrará el *bean* de la clase falsa (*ArticulosEntidadServicioValidacion**Fake***) en vez del real que contiene todas las validaciones.

A este código faltaría llamarlo desde el test creado anteriormente de la siguiente forma:

```
@BeforeEach
public void antesDeCadaTest() throws Exception {
    deshabilitarValidacionesExcepto("tiposArticuloEntidadServicioValidacionImpl");
}
```

Figura 66. *Llamada antes de cada test a la desactivación de las validaciones*

Es por esto por lo que, gracias a esta implementación, se desactivarán las validaciones de ARTICULOS y se mantendrán activas las de TIPOS_ARTICULO.

8. Conclusiones y trabajos futuros

Llegados al final, es hora de resumir lo más destacable. Partiendo de los objetivos establecidos y los conceptos que se pretendían enseñar, se sintetizará lo que se ha aprendido junto con una reflexión sobre lo que ha aportado el desarrollo de esta memoria tanto personal como profesionalmente.

Además de esto, sugeriremos un camino a seguir para continuar aprendiendo acerca de esta tecnología tan demandada. Proponiendo ideas para lograr nuevos objetivos en base a los ya obtenidos.

8.1 Conclusiones

La filosofía con la que más se identifica esta memoria se resume, de manera precisa, con una cita del famoso matemático Alan Mathison Turing: "solo podemos ver poco del futuro, pero lo suficiente para darnos cuenta de que hay mucho que hacer". Y es que, redactando esta memoria, se ha tenido muy presente la sensación de que es una tarea compleja intentar resumir lo más importante de un framework tan potente. A su vez, el poder compartir conocimiento y que quede plasmado otorga una sensación de plenitud muy gratificante. Especialmente, si se consiguen todos nuestros objetivos.

A lo largo de la memoria, se han presentado las tareas más útiles, prácticas y reales que todo programador backend en Spring debe conocer. No sin antes conocer las bases del framework, como lo son el patrón MVC y la inyección de dependencias. Estas características tan peculiares de Spring, las bases que ya poseíamos de la universidad y la investigación personal han dado como resultado el poder mostrar la magnificencia de este framework.

Aunque quedan un gran número de detalles por aprender sobre esta tecnología, gracias a esta memoria hemos podido resaltar la importancia de las ventajas que nos ofrece. Mostrando de las partes más complejas a las más sencillas, de las más demandadas a las que conllevan cierto grado de imaginación. Nos referimos a esto, entendiendo por complejas al servicio REST, por sencillas a la adición de los atributos de Thymeleaf, por demandadas al servicio REST incorporando llamadas a la lógica de la capa de persistencia junto con un *gateway* y por creativas a la implementación del Eureka Server.

En cuanto a las tecnologías presentadas. Es cierto que no se ha mostrado el potencial que poseen tanto como sería beneficioso, pero si es correcto hacer hincapié de que sirven de gran utilidad en un ámbito laboral. Aunque no se aprecie en la memoria, el no usar Git en la vida laboral es inconcebible. La combinación de los IDE de base de datos, mencionados en el apartado de Estado del arte, permiten una gestión de la capa de persistencia que roza la perfección. El motivo por el cual no se ha detallado su uso es debido a la facilidad con la que se aprenden. Además de que la prioridad siempre ha sido centrarse en Spring.

La importancia de Hibernate junto a JPA, el lenguaje PL/SQL, las propiedades básicas de todo proyecto Spring, la diferencia entre controladores, servicios, entidades extendidas, super clases, *fixtures*, repositorios, etcétera es solo el inicio de un camino donde la única barrera es la limitación personal que cada uno anteponga a sus ganas de aprender.

Spring con a sus anotaciones y la creación de Spring Boot, nos suministra artefactos que permiten crear proyectos en cuestión de minutos completamente utilizables sin apenas configuración. Compaginando estos proyectos con la posibilidad de realizar validaciones, el efecto que provoca en uno mismo es tener esa seguridad de que si algún día necesitas implementar un backend, siempre se tendrá en mente Spring. Porque ésta, es la clave de Spring.

8.2 Trabajos futuros

No podemos despedirnos de la memoria sin conocer cuáles son los siguientes pasos que recomendamos para seguir completando nuestra formación:

- **Automatización de la estructura de test con Gradle**

Como se mencionó en el apartado de Retos principales, la estructura que sigue la creación de test que compruebe el correcto funcionamiento de las validaciones es siempre la misma. Primero la entidad, luego el repositorio, seguido del servicio de validaciones inyectando su dependencia en la entidad, junto con el fixture y, por último, el test. Es por este motivo, que se presenta el reto de crear esta estructura usando tareas Gradle que lo automaticen.

El objetivo es que, ejecutando la tarea por la línea de comandos, cree los ficheros correspondientes. Las tareas Gradle pueden crearse individualmente, como veremos en la imagen, o una que englobe al resto:

Figura 67. *Ejemplo de tareas Gradle para automatizar*

Para crearlas será necesario:

1. Crear un fichero llamado, por ejemplo: *tareasTFG.gradle* donde se encontrará la implementación.
2. Dentro de *build.gradle* añadir la dependencia con: "*apply from: 'tareasTFG.gradle'*".
3. Para llamarlas por la línea de comandos de IntelliJ será necesario escribir:

```
gradlew generarFixture -P tabla=prueba
```

Mediante `-P propiedad=valor` realizaremos el paso de parámetros que recogeremos en la implementación.

La proceso sería: asignar la tarea al grupo, recibir el valor de las propiedades, declarar entre las variables locales la ubicación donde se encuentra el archivo que se va a crear y el construir el nombre de las variables en función de las enviadas como parámetros, crear el archivo y añadir su contenido variable. Lo que se encuentra entre "[…]" es porque es un path y es variable.

```
task generarFixture {
   setGroup('tfg')
   def pathFixtures = 'src/test/java/[…]/fixtures/'
   def entity = project.getProperties().get('tabla').toString() // prueba
   def nombreClase = entity.toString().capitalize() + 'Fixture'
      // PruebaFixture
   def nombreArchivo = nombreClase + '.java' // PruebaFixture.java
   def qualifier = '"' + entity + 'Fixture' + '"' // "pruebaFixture"

   File archivo = new File(pathFixtures + nombreArchivo)
   doFirst {
      if (!archivo.exists()) {
         archivo.append("package […].fixtures;\n\nimport
org.springframework.stereotype.Service;\n\n@Service(" + qualifier + ")\npublic
class " + nombreClase + "{\n}")

         archivo.createNewFile()
      }
   }

   doLast {
      if (archivo.exists()) {
         println 'Fixture: ' + nombreClase + ' creado en la ruta: ' +
pathFixtures
      }
   }
}
```

Figura 68. *Implementación de tarea Gradle*

El resultado obtenido sería el haber construido el archivo con la cabecera de nuestro *fixture* relleno:

```
package […].fixtures;

import org.springframework.stereotype.Service;

@Service("pruebaFixture")
public class PruebaFixture {
}
```

Figura 69. *Resultado de la ejecución de la tarea Gradle*

Si llevamos esto al extremo y concatenamos la creación de todos los archivos, nos aseguraríamos de que se crean en la carpeta correspondiente de forma automática, con el contenido variable ya implementado y solo tendríamos que añadir la lógica de las validaciones y el test.

Con esta idea lo que se pretende, aparte de mostrar como automatizar la creación de la estructura de test, es enseñar un uso personalizado de las tareas Gradle y su potencial. Se deja en manos del programador jugar con su creatividad y añadirle otro tipo de valor a esta herramienta.

- **Microservicio generador de super clases**

La idea principal es desarrollar un microservicio que genere automáticamente las super clases Java en base a las tablas que haya en la base de datos. Es decir, si creamos una tabla mediante SQL y ejecutamos nuestro futuro microservicio, el resultado que debe devolvernos es el haber creado la super clase con las anotaciones Spring teniendo en cuenta las columnas, restricciones, etc.

Es cierto que las relaciones sería el apartado más difícil de implementar, pero no somos partidarios de limitar lo que se puede llevar a cabo o no en el mundo de la programación.

- **Reactivar microservicios**

Dado que Eureka Server registra microservicios y nos muestra su estado, sería ideal poder desarrollar una funcionalidad que reactive un microservicio una vez terminase su ejecución. En el ámbito laboral, permitiría que una empresa recibiese menos incidencias por cuestiones relacionadas con la caída de un microservicio.

- **Servicio de base de datos general**

Teniendo en cuenta que para insertar, actualizar o eliminar registros de la base de datos debemos inyectar la dependencia correspondiente de cada repositorio perteneciente a una entidad particular, podríamos hacer un servicio Java que realizase estas operaciones de manera general a todas las entidades. Dicho en otras palabras, se encomienda crear un método para insertar que actualizase si ya existe y otro para eliminar, que recibiesen como parámetro un objeto de cualquier entidad y realizase la misma operación que su repositorio.

- **Integración continua o CI**

Aunque esta idea es la menos original, debido a que su popularidad está presente en la mayoría de las empresas, siempre viene bien recordar que se podría implementar un proceso que compile el código y ejecute los test. En caso de sean superados con éxito, el proceso continuaría realizando una fusión de las ramas añadiendo los cambios en la rama master.

9. Referencias

[1] Spring Framework: Qué es y por qué usarlo. (2020, 26 noviembre).

https://ifgeekthen.everis.com/es/spring-framework

[2] García, D. (2014, 17 enero). Inversión de Control e Inyección de Dependencias.

https://danielggarcia.wordpress.com/2014/01/15/inversion-de-control-e-inyeccion-de-dependencias/

[3] O. (2016, 1 diciembre). El concepto Inversion of Control.

https://www.oscarblancarteblog.com/2016/12/01/concepto-inversion-of-control/

[4] A. (2019, 9 diciembre). Diferencias entre Spring y Spring Boot.

https://javadesde0.com/diferencias-entre-spring-y-spring-boot/

[5] (S/f). Spring.io.

https://start.spring.io/

[6] Atlassian. (s. f.). Qué es Git: conviértete en todo un experto en Git con esta guía.

https://www.atlassian.com/es/git/tutorials/what-is-git

[7] Fork - a fast and friendly git client for Mac and Windows. (s/f). Git-fork.com.

https://git-fork.com/

[8] Manual de desarrollo en PL/SQL | Marco de Desarrollo de la Junta de Andalucía. (s.f.).

http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/107#:%7E:text=PL%2FSQL%20es%20el%20lenguaje,Funciones

[9] SqlDbx. (s/f). Sqldbx.com.

http://www.sqldbx.com/

[10] ¿Qué es Java Hibernate? ¿Por qué usarlo? (2021, 15 abril).

https://ifgeekthen.everis.com/es/que-es-java-hibernate-por-que-usarlo

[11] ¿Cuál es la diferencia entre JPA e Hibernate? [cerrado]. (s. f.).

https://qastack.mx/programming/9881611/whats-the-difference-between-jpa-and-hibernate

[12] Qué es MVC. (2020, 28 julio).

https://desarrolloweb.com/articulos/que-es-mvc.html

[13] Gamma, E. (2002). *Patrones de diseño* (1.a ed.). Madrid, España: Pearson Educación.

[14] Salas, I. (2018, 12 septiembre). Inyección de dependencias en Spring.

https://programandoointentandolo.com/2013/05/inyeccion-de-dependencias-en-spring.html

[15] API Gateway. (s. f.).

https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/api-gateway

[16] JPA - Relaciones de la Entidad. (s. f.).

https://www.tutorialspoint.com/es/jpa/jpa_entity_relationships.htm

[17] Spring Data JPA - Reference Documentation. (s. f.).

https://docs.spring.io/spring-data/jpa/docs/1.4.3.RELEASE/reference/htmlsingle/

[18] Rodríguez, S. C. (s. f.). Thymeleaf como alternativa MVC.

https://blog.softtek.com/es/thymeleaf

[19] Fernández, D. (2021, 13 abril).
Arquitecturas basadas en microservicios: Spring Cloud Netflix Eureka.
https://blog.bi-geek.com/arquitecturas-spring-cloud-netflix-eureka/

[20] Yuste, M. D. (2020, 7 marzo). Spring Cloud Series:
Crea un Servicio de Registro y Descubrimiento con Spring Cloud Netflix Eureka paso a paso.
https://migueldoctor.medium.com/spring-cloud-series-crea-un-servicio-de-registro-y-descubrimiento-con-spring-cloud-netflix-eureka-4758615ad4cb

[21] Christudas, B. (2019). *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. California, EEUU: Apress Media.