Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

TRANSICIÓN DEL MONOLITO A LOS MICROSERVICIOS: CAMBIO DE PARADIGMA EN EL DESARROLLO DE APLICACIONES

Autora
Ana Huerta Tajuelo

Directora
Noelia Quintana Saavedra

Madrid

Transición del monolito a los microservicios: cambio de paradigma en el desarrollo de aplicaciones

Autora: Huerta Tajuelo, Ana.
Directora: Quintana Saavedra, Noelia.

Palabras clave: microservicios, monolito, arquitectura, desarrollo de aplicaciones, software, Spring Boot, servicio REST

Resumen del Proyecto

El presente proyecto estudia el proceso de transformación de una arquitectura monolítica a una de microservicios. Su objetivo principal es el estudio exhaustivo de las arquitecturas de microservicios, centrándose en las diferencias de estas respecto a las arquitecturas monolíticas. Dichas diferencias se exponen junto al desarrollo de una aplicación en la que se aprecia la transición de un tipo de arquitectura al otro, con la esperanza de que pueda servir como guía para futuros curiosos, estudiantes o desarrolladores.

Estado de la cuestión

Para entender la aparición de los microservicios, es indispensable revisar qué son, qué están reemplazando y por qué son necesarios.

En la década de los 80 comienzan distintos intentos de simplificar las llamadas a procedimientos remotos. Si se lograba este supuesto, sería posible crear sistemas constituidos en varias máquinas evitando los problemas de memoria y escalabilidad de la época.

Tras varias iniciativas poco exitosas, a mediados de la década de los 2000 nace lo que ahora se conoce como Arquitectura Orientada a Servicios (SOA, del inglés *Service Oriented Architecure*). SOA, a su vez, surge del *Simple Object Access Protocol* o SOAP, de Microsoft, cuya idea fundamental es hacer el software más sencillo posible, y que consistía en invocar métodos a través de HTTP. SOAP demostró que era posible operar entre sistemas desarrollados en diferentes lenguajes y plataformas.

Sin embargo, SOA añadió numerosas capas al protocolo SOAP, que no es considerado sencillo, lo cual hizo resurgir el problema inicial: la dificultad para implementar llamadas remotas.

Con ello, la industria se alejó de SOAP en favor de *Representational State Transfer* o REST, proveniente de la tesis doctoral de Roy Fielding en el año 2000 [1], que se define como una interfaz entre sistemas que usa HTTP para obtener datos u operar sobre datos [2]. Además, proporciona una nueva manera de especificar

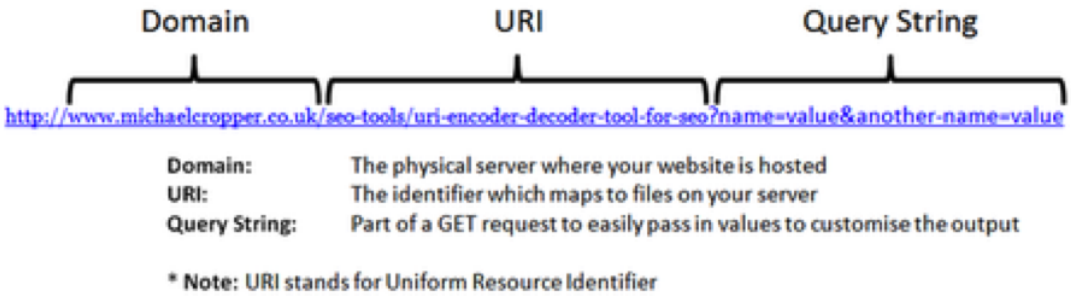


Figura 1: Esquema de la estructura de URIs

nombres de entidades utilizando URIs, cuya estructura se muestra en la siguiente figura.

La popularidad de los servicios REST allanó el terreno para los microservicios, que requieren de comunicación rápida, fiable y eficiente. La arquitectura de microservicios comenzó a generar expectación tras los conocidos éxitos de Netflix o Amazon. Sin embargo, estas compañías y muchas otras que han conseguido grandes resultados con microservicios tienen una cosa en común: todas son recientes y desarrollaron aplicaciones nuevas. Ninguna de ellas tenía un proyecto en marcha que debiera transformar o reemplazar [1]. Para empresas tradicionales, que existen desde hace años, embarcarse en este modelo cuenta con la complicación de tener que refactorizar una aplicación monolítica de una envergadura considerable. Sin embargo, en muchas ocasiones son precisamente estas aplicaciones las que más se beneficiarían de una transición a los microservicios [1].

Descripción del proyecto

Para ayudar a difundir el conocimiento de dicha transformación, el proyecto consistirá en la implementación de cuatro aplicaciones REST completas. Cada una de ellas constituirá una tienda de venta *online* de componentes electrónicos. Sin embargo, tendrán características y estructuras diferentes y emplearán una variedad de tecnologías. Dichas aplicaciones serán las encargadas de mostrar las diferencias entre ambas arquitecturas.

La primera de ellas será una sencilla aplicación monolítica, que ni siquiera constará de base de datos. La segunda aplicación surge de la evolución de la anterior al incorporar una base de datos mySQL, a la que se conectará mediante un controlador JDBC. Es también, por tanto, una aplicación monolítica.

En la tercera aplicación acontece el cambio a una arquitectura de microservicios. Este hito, además, da paso al uso de tecnologías más actuales como JPA, Hibernate, H2 y Spring Data, que suponen un cambio radical en el tratamiento de

datos.

Finalmente, la cuarta aplicación contará con la adición de un segundo microservicio, que reforzará la comprensión de la arquitectura y evidenciará su sencilla escalabilidad.

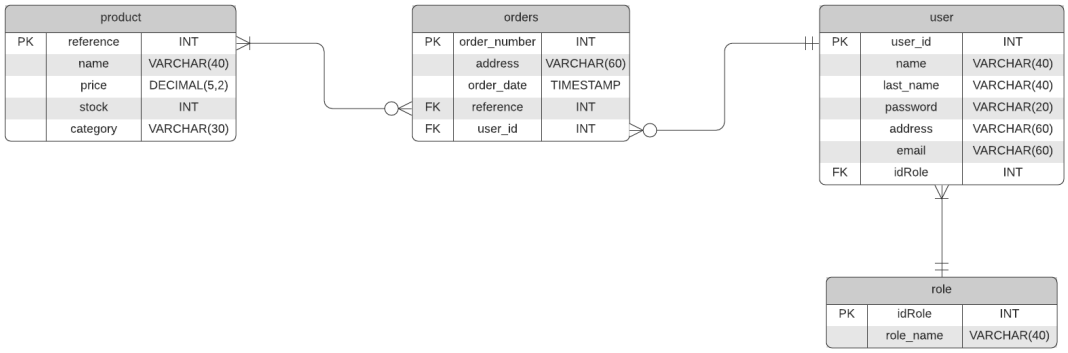


Figura 2: Modelado de la base de datos del sistema

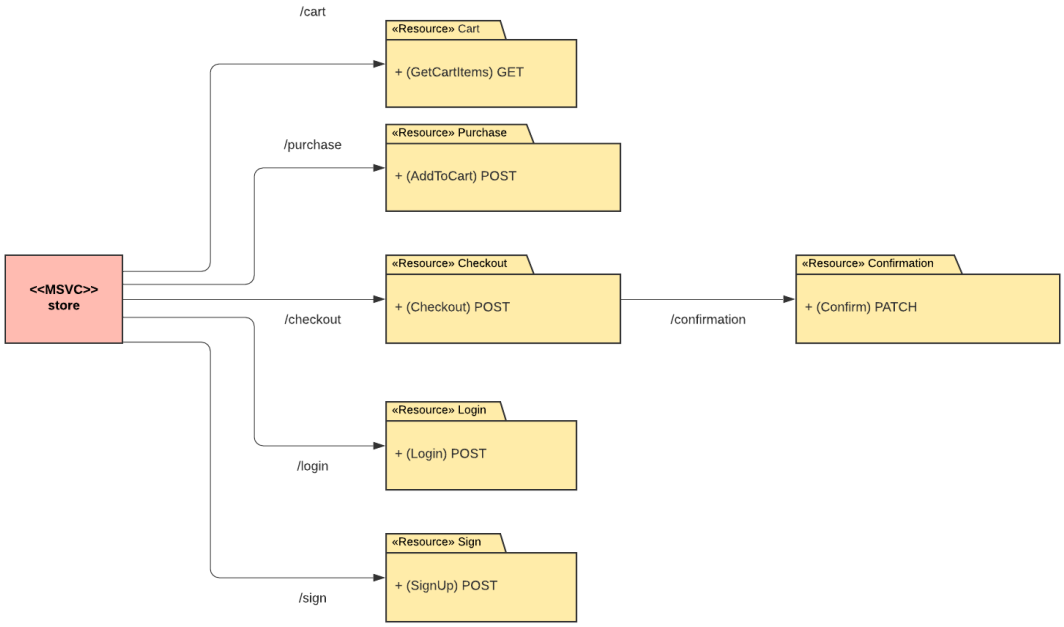


Figura 3: Esquema del servicio REST

Resultados

Debido al desarrollo de cuatro aplicaciones, las diferencias entre ellas quedan más localizadas y delimitadas, facilitando así la comprensión de su finalidad y funcionamiento.

La aplicación inicial funciona correctamente. Es la que cuenta con más limitaciones, dada la sencillez intencionada de su diseño. Se inicializan varios productos, que son inmutables excepto por su *stock*. Tampoco contiene registro de usuarios.

Asimismo, la segunda aplicación cumple su cometido. Tras definir y crear las tablas *orders*, *product* y *user*, e insertar algunos registros en ellas, la aplicación está capacitada para que los usuarios ya existentes realicen compras. No cuenta con funcionalidad para permitir a nuevos usuarios darse de alta.

En la tercera aplicación se efectúa el cambio de arquitectura. Se observan diferencias significativas con la anterior, siendo estas estructurales, de rendimiento, de alcance y, sobre todo, de las tecnologías empleadas para su desarrollo. La aplicación funciona según lo previsto, y extiende la funcionalidad de su predecesora, permitiendo la incorporación de nuevos clientes.

Por último, la aplicación más completa representa la evolución natural de la anterior, extendiendo su complejidad y capacidad. Desempeña el mayor número de tareas de todas ellas, y lo hace de forma satisfactoria.

Dados los resultados obtenidos, se confirma que el proyecto cumple con los objetivos descritos en su proposición.

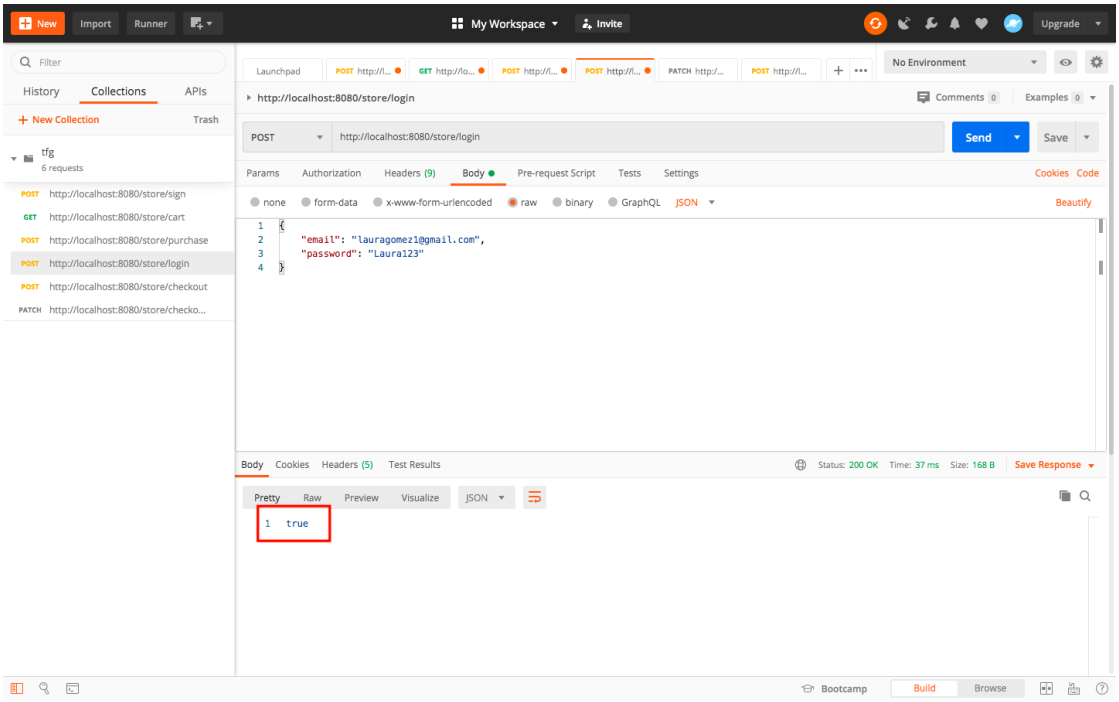


Figura 4: Ejemplo de una de las funciones de la aplicación

The monolith to microservices transition: a paradigm shift in software development

Ana Huerta Tajuelo

Keywords: microservices, monolith, architecture, software development, Spring Boot, REST service

Abstract

This project studies the process of transforming a monolithic architecture into a microservices based one. The project's primary goal is the exhaustive study of microservices, focusing on their differences from monoliths. Those dissimilarities are presented throughout the development of an application that displays the transition from an architecture to the other, hoping to be of use to future aficionados, students and developers.

State of the question

To understand the emergence of microservices, it is crucial to review what they are, what they are replacing and why they are paramount.

The decade of the 1980s saw the rise of several attempts to simplify the calls to remote procedures. If this was achieved, it would become possible to create systems built in several machines, avoiding the memory and scalability issues of the time.

After several efforts found little success, in the mid-2000s the Service Oriented Architecture - or SOA - is born. SOA arises from Microsoft's SOAP (Simple Object Access Protocol), whose core idea was to build the simplest software possible, which it did by invoking methods through HTTP. SOAP demonstrated it was possible to operate between different systems, languages and platforms.

However, SOA added may layers to SOAP, which was already a complicated protocol. This resurfaced the initial problem: the difficulty of implementing remote calls.

That made the industry drop SOAP in favor of REST, which stands for Representational State Transfer and was presented in Roy Fielding's doctoral thesis in 2000 [1]. It is defined as an interface between systems that uses HTTP to obtain or modify data [2]. Furthermore, it provides a new way of naming entities by using Universal Resource Identifiers. A URI's structure is presented in the following figure.

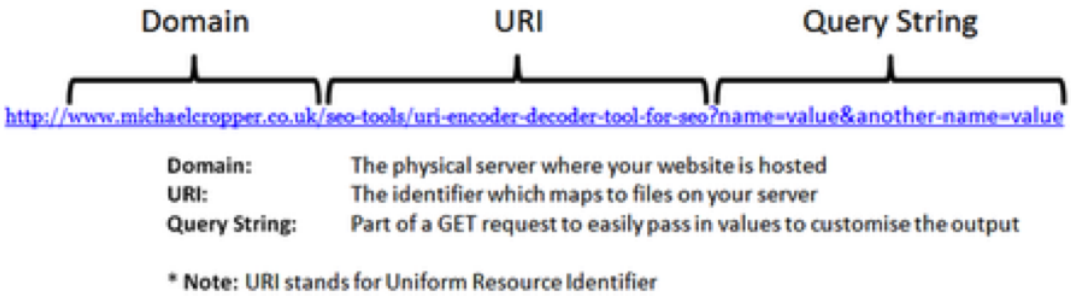


Figura 5: Structure of a URI

The popularity of REST services eased the arrival of microservices, which require fast, reliable and efficient communication. The microservices arquitecture started to gain momentum after the renowned success of both Netflix and Amazon. However, these companies and many others that have benefitted from microservices have a thing in common: they are all new companies and developed their applications from scratch. None of them had an ongoing project that had to be transformed or replaced [1]. For established companies that have existed for a number of years, getting involved in this model would mean undergoing a transformative process of their current application. However, more often than not, this kind of companies with a massive application are the ones that would benefit the most from switching to microservices [1].

System description

In order to promulgate information about that transformation, the project will implement four REST applications. Each of them will comprise an electronic components online store. Nonetheless, they will comprehend different characteristics, structures and technologies. These applications will be in charge of demonstrating the contrast between both architectures.

The first of them is a simple, straightforward monolith which does not have a database. The following application comes from its evolution after including a mySQL database and establishing a connection through a JDBC driver. It is considered a monolithic application as well.

The shift to a microservices architecture takes place in the third application. This milestone is accompanied by the introduction of more modern technologies into the project, such as JPA, Hibernate, H2 and Spring Data, which completely transform data processing in the application.

Ultimately, the fourth application features a second microservice, which will reinforce the understanding of the project, as well as showcase how scalable mi-

croservice architectures are.

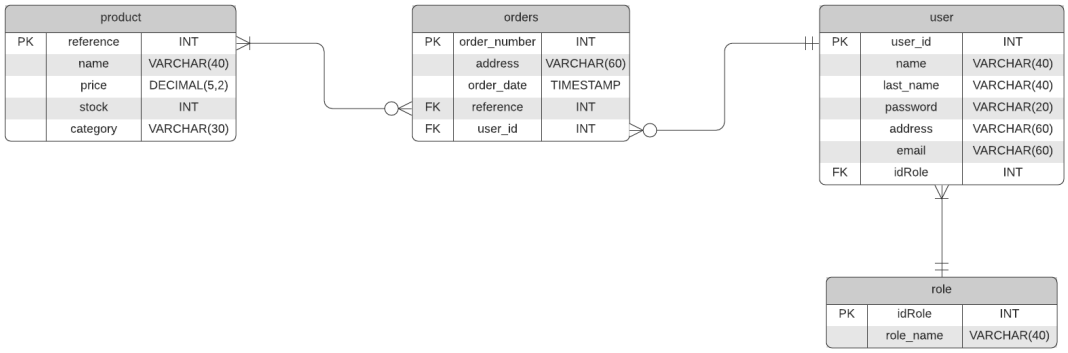


Figura 6: Application's database design

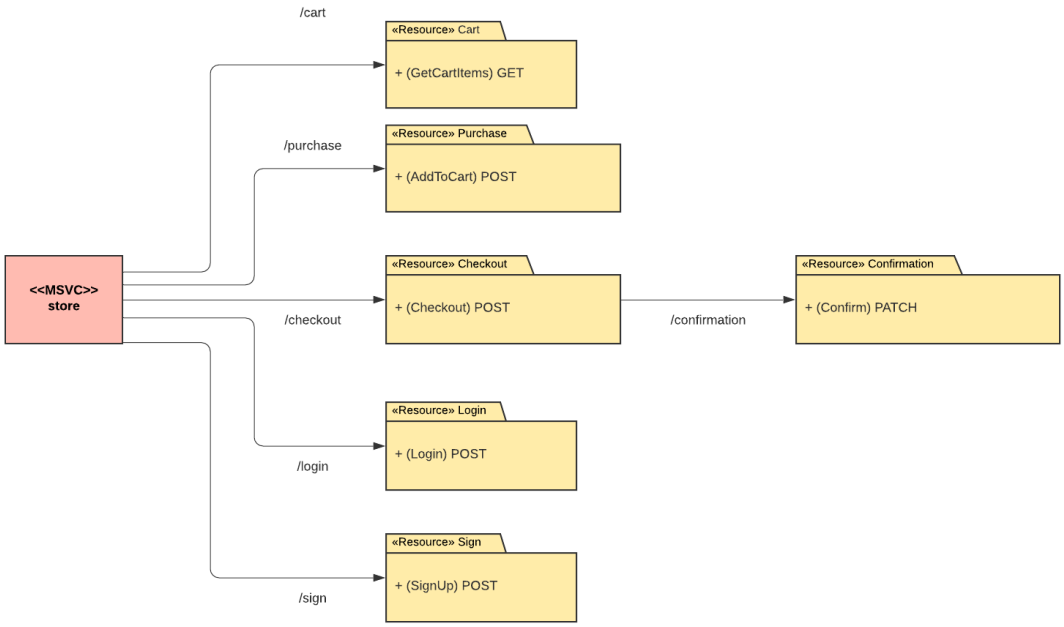


Figura 7: REST API structure

Results The development of four applications allows for a clear location of the differences amongst them, which, in turn, provides an opportunity for better understanding of their purpose and functioning.

The initial application functions properly. It is the simplest and most limited

of them, as it was intended to be. Several products are initialized, and they are immutable except for their stock. It also lacks a user record.

Likewise, the second application performs as expected. After establishing and creating the tables orders, product and user, and inserting several records in them, the application is capable of having existing users place orders. It is, however, not suited to let new users sign in.

The third application follows a microservice pattern. There are significant alterations between the second and third versions, including scope, performance and structural changes. The most compelling difference is, however, the tools and frameworks handled in its development. The application works according to plan and extends the functionality of its predecessor, allowing new customers to sign in.

Lastly, the most complete application is the natural evolution of the one discussed above, extending its complexity and capabilities. It performs the highest amount of tasks out of the four versions, and it does so in a satisfactory manner.

Given the results showcased above, the project fulfills the goals presented at its inception.

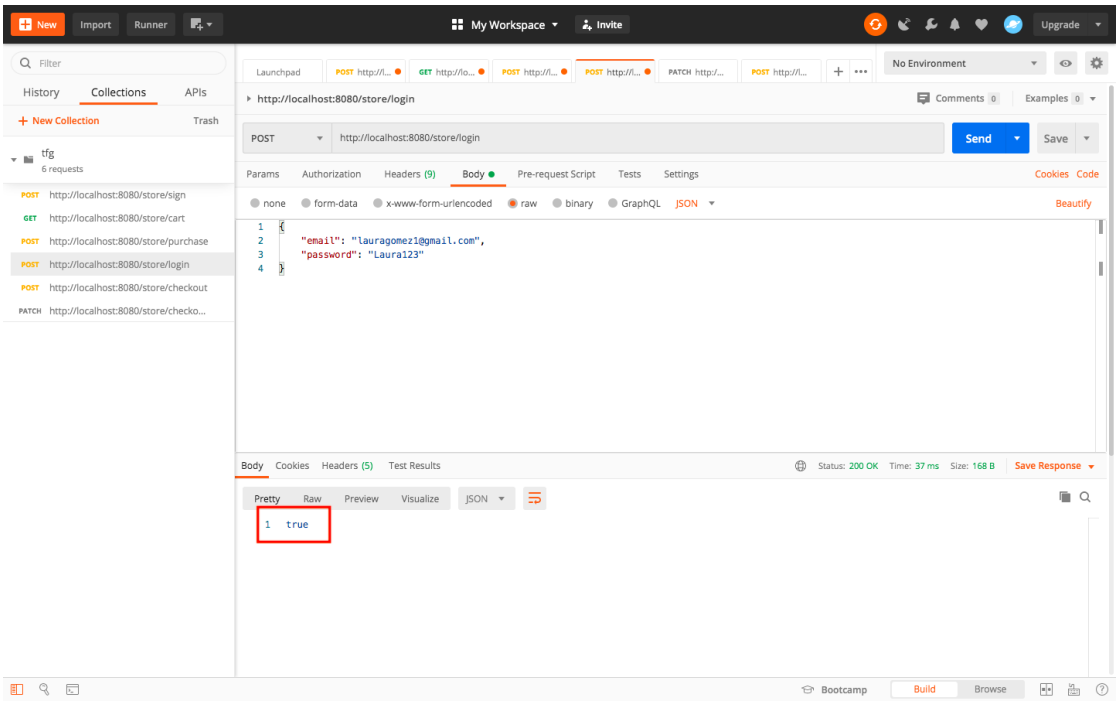


Figura 8: Example from one of the application's tasks

Bibliografía

[1] Brown, K. *Beyond buzzwords: A brief history of microservices patterns.* Disponible en https://developer.ibm.com/technologies/microservices/articles/cl-evolution- microservices-patterns/

[2] BBVA Open4U. *API REST: qué es y cuáles son sus ventajas en el desarrollo de proyectos.* Disponible en https://bbvaopen4u.com/es/actualidad/api-rest-que-es- y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Estado de la cuestión

Nos encontramos en la cuarta era del *software*, en la que el *hardware* con menor vida útil y el *software* de aplicación se han convertido en productos básicos. Es la era de transición de aplicaciones *web* a dispositivos, de redes cableadas a redes de fibra óptica e inalámbricas, y de un modelo de *software* basado en aplicaciones a uno basado en servicios [1].

Dichos cambios, sumados a la democratización de la tecnología y a las crecientes expectativas de los usuarios, han supuesto un incremento en la complejidad y envergadura de las aplicaciones. El diseño y organización de la totalidad del sistema ha pasado a ser una de las tareas más delicadas e importantes en la creación de una aplicación.

Aparecen también nuevas formas de trabajar, motivadas por los cambios sociales además de los tecnológicos. La demanda actual de agilidad y capacidad de respuesta ha dejado obsoleta la idea de producir *software* de la misma manera que se fabrica un coche, pasando por fases de planificación, diseño, arquitectura, desarrollo, pruebas y lanzamiento. El desarrollo de una parte del *software* no espera a que las anteriores estén listas [2]. Esto permite que múltiples equipos, situados incluso en distintos lugares del mundo, puedan trabajar simultáneamente en una misma aplicación. Este nuevo planteamiento es poco compatible con la tradicional arquitectura monolítica, en la que se produce una pieza esencial de *software*, con componentes interconectados e interdependientes, y que requiere que todos ellos sean compilados y ejecutados juntos [3]. La arquitectura de microservicios aparece como posible solución a esta cuestión.

El término "microservicios" aparece por primera vez en un taller para arquitectos de *software* en mayo de 2011. Con él, se quería denominar un tipo de arquitectura con el que muchos de ellos experimentaban en aquel momento. En mayo

de 2012, el mismo grupo de personas acuña de forma definitiva este término [4].

Para entender la aparición de los microservicios, es necesario revisar qué son, qué están reemplazando y por qué son necesarios.

En la década de los 80 comienzan distintos intentos de simplificar las llamadas a procedimientos remotos. Si se lograba este supuesto, sería posible crear sistemas constituidos en varias máquinas evitando los problemas de memoria y escalabilidad de la época. Tras iniciativas poco exitosas tales como CORBA o Facade, a mediados de la década de los 2000 nace lo que ahora se conoce como Arquitectura Orientada a Servicios (SOA, del inglés *Service Oriented Architecure*). SOA, a su vez, surge del *Simple Object Access Protocol* o SOAP, de Microsoft, cuya idea fundamental es hacer el *software* más sencillo posible, y que consistía en invocar métodos a través de HTTP [5].

Por su parte, HTTP (*Hypertext Transfer Protocol* o Protocolo de Transferencia de Hipertexto en español) es un protocolo de la capa de aplicación que fue diseñado para la comunicación entre navegadores y servidores *web*. Se trata de un protocolo sin estado, lo que significa que el servidor no guarda ningún dato (estado) entre dos peticiones [6]. El hecho de que HTTP sea *stateless* será determinante en la aparición de un enfoque presentado más adelante. Cada petición HTTP está encabezada por un verbo, que le indica al servidor qué hacer con los datos recogidos de la URL (GET, DELETE, etcétera).

SOAP demostró que era posible operar entre sistemas desarrollados en diferentes lenguajes y plataformas. Sin embargo, SOA añadió numerosas capas a un protocolo que es considerado complejo de por sí, lo cual hizo resurgir el problema inicial: la dificultad para implementar llamadas remotas.

Con ello, la industria se alejó de SOAP en favor de *Representational State Transfer* o REST, proveniente de la tesis doctoral de Roy Fielding en el año 2000 [5]. Este se define como una interfaz entre sistemas que usa HTTP para obtener datos u operar sobre datos [7]. Al utilizarla, una aplicación puede interactuar con un servicio conociendo tan solo el identificador del servicio y la acción que se debe realizar. Aquí entran en juego los verbos HTTP mencionados anteriormente, que REST utiliza para especificar dicha acción. En RESTful APIs, los más empleados como norma general son GET, POST, PUT y DELETE, pero existen muchos otros.

Los principios de una arquitectura de estilo REST son los siguientes:

- Separación del cliente y el servidor, de forma que mejore la portabilidad de la interfaz de usuario a diferentes plataformas, así como la escalabilidad de la aplicación al simplificar los componentes del servidor.

- *Stateless*, concepto introducido anteriormente al hablar de HTTP. Antepone que cada petición del cliente al servidor debe contener toda la información

necesaria para entender dicha petición y, por tanto, no se empleará ningún dato almacenado en el servidor.

- Las respuestas a peticiones deberán ser etiquetadas como cacheables o no cacheables. En el primero caso, se podrá reutilizar la información de esa respuesta para peticiones equivalentes. En caso contrario, no podrá almacenarse dicha información.

- Sistema de capas, que permite limitar el comportamiento de los componentes al impedir que "vean"más allá de la capa con la que interactúan.

- Código bajo demanda, que extiende la funcionalidad del cliente al descargar y ejecutar código en forma de *applets* o *scripts*. Esto simplifica el cliente mediante la reducción del número de funciones que deben ser pre-implementadas. Permitir que se descarguen tras el despliegue mejora la extensibilidad del sistema.

- Interfaz común entre componentes [8]. Todos los recursos se identifican mediante URIs, del inglés *Uniform Resource Identifier*, que pueden determinar, por ejemplo, tanto una página web como al remitente o al destinatario de un correo electrónico. Es la forma en la que el sistema identifica a qué información debe acceder, dónde y cómo [9]. La siguiente figura muestra un esquema de su estructura.

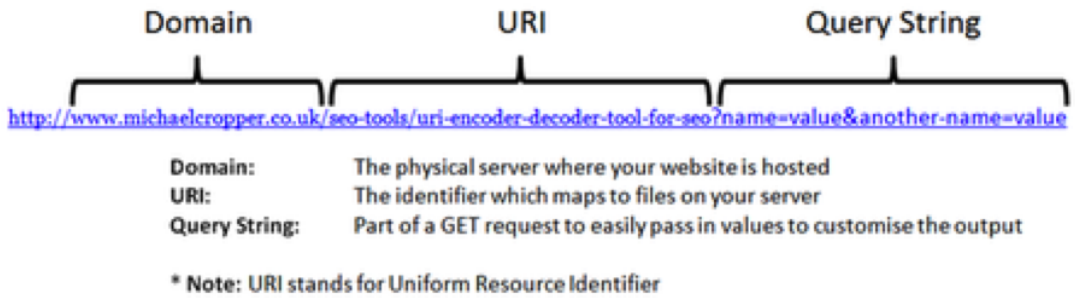


Figura 1.1: Estructura de un URI

Las mejoras introducidas por los servicios REST allanaron el terreno para la aparición de los microservicios. Estos requieren de comunicación rápida, fiable y eficiente, que los predecesores de REST no eran capaces de proporcionar.

La arquitectura de microservicios es un tipo de estructuración emergente en el desarrollo de *software*. Su idea fundamental es la separación de los diversos servicios que compondrían una aplicación, para así poder desplegarlos, modificarlos o ampliarlos cuando se desee, sin interferir con el resto. Por esta razón, la arquitectura de microservicios tiene una escalabilidad muy sencilla.

En las aplicaciones monolíticas todos los cambios están vinculados unos a otros y, por tanto, cualquier modificación en una sección convierte la aplicación en una versión completamente nueva, con lo que ello conlleva para su despliegue.

A pesar de que no hay un estándar definido para la arquitectura de microservicios, se observan algunas características comunes:

- El *software* de una aplicación de microservicios se puede descomponer en distintas partes funcionales independientes. Esto es, cada microservicio podrá ser desplegado, modificado y desplegado de nuevo sin comprometer el resto de la aplicación. Por tanto, el caso descrito anteriormente en el que la aplicación se convierte en una nueva versión a desplegar no se da con microservicios.

- Los microservicios suelen organizarse en torno las necesidades y prioridades del cliente. En la arquitectura de microservicios los equipos trabajan en módulos multifuncionales que realizan una tarea específica, mientras que en las monolíticas cada equipo tiene un enfoque específico sobre una parte de la aplicación. Esto supone un gran ahorro en tiempo de desarrollo, además de brindar la posibilidad de realizar tareas de mantenimiento en una parte de la aplicación sin interrumpir al resto de equipos.

- Mientras que los entornos ESB (*Enterprise Service Buses*) clásicos utilizan equipos para enrutar mensajes, redireccionar tráfico, denegar ciertos accesos, etcétera, los microservicios se basan en la idea de recibir una petición, procesarla y generar una respuesta consecuente.

- En una arquitectura de microservicios, cada módulo contará con su propia base de datos. La ventaja de esto es eliminar el riesgo de sobrecargarla con numerosas solicitudes de distintas funcionalidades de la aplicación y provocando el fallo de la aplicación completa.

- Cuando se cuenta con varios microservicios, estos suelen estar comunicados entre sí. Se implanta un sistema de aviso y actuación en caso de que alguno de ellos falle, filtrando adecuadamente la información destinada a este módulo y favoreciendo la correcta gestión de recursos en los módulos restantes [10].

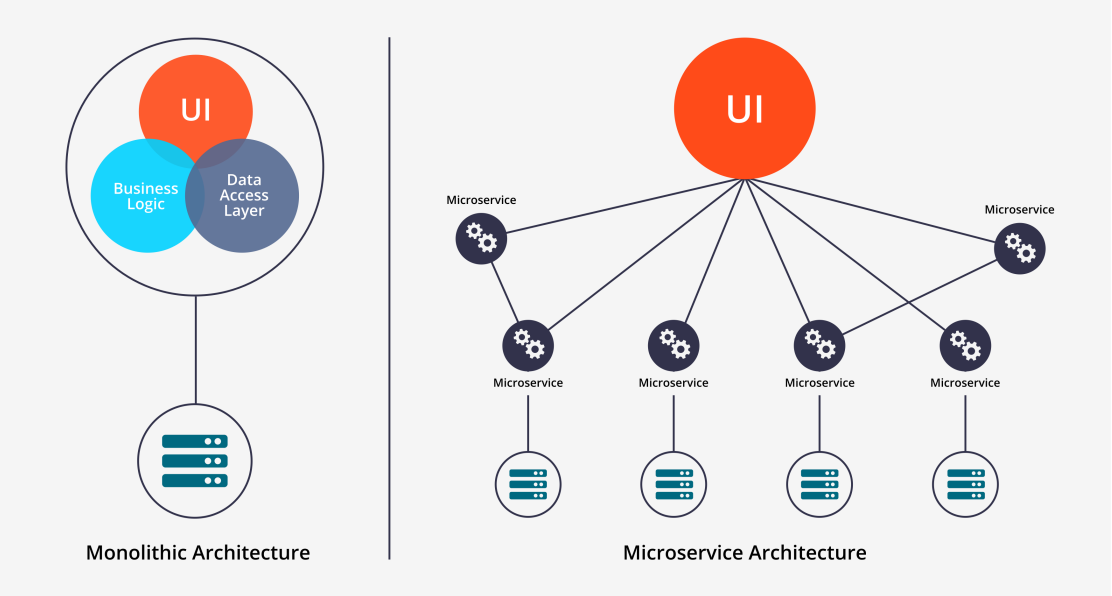


Figura 1.2: Comparación esquemática de ambas arquitecturas.

Tras estudiar las características de los microservicios, se listan algunas de sus ventajas:

- Agilidad. Dado que se implementan de manera independiente, resulta más fácil administrar las correcciones de errores y las versiones de la aplicación. Se puede actualizar un servicio sin volver a desplegar toda la aplicación, además de revertir una actualización si algo va mal. En muchas aplicaciones tradicionales, un error en una parte de la aplicación puede bloquear todo el proceso de lanzamiento. Es posible que se requieran nuevas características a la espera de que se integre, pruebe y publique una corrección de errores.

- Equipos pequeños y centrados. Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo lo pueda compilar, probar e implementar. Los equipos pequeños favorecen la agilidad. Los equipos grandes suelen ser menos productivos, porque la comunicación es más lenta, aumenta la sobrecarga de trabajo administrativo y la agilidad disminuye.

- Base de código pequeña. En las aplicaciones monolíticas, con el paso del tiempo aparece la problemática de que las dependencias de código terminen por enredarse, por lo que para agregar una nueva característica, es preciso tocar el código en muchos puntos. Al no compartir el código ni las bases de de datos, la arquitectura de microservicios minimiza las dependencias y resulta más fácil agregar nuevas características.

- Mezcla de tecnologías. Los equipos pueden elegir la tecnología que mejor se adapte al servicio sin que esto interfiera con el resto de la aplicación.
- Aislamiento de errores. Si un microservicio individual no está disponible, no interrumpe toda la aplicación, siempre que los microservicios de nivel superior estén diseñados para controlar los errores correctamente.
- Escalabilidad. Los servicios se pueden escalar de forma independiente. Esto permite escalar horizontalmente los subsistemas cuando requieran más recursos, sin tener que escalar horizontalmente toda la aplicación [11].

Por otro lado, los desafíos que se presentan al trabajar con esta arquitectura son los siguientes:

- Complejidad. Una aplicación de microservicios tiene más partes que la aplicación monolítica equivalente. Cada servicio es más sencillo, pero el sistema en conjunto es más complejo.
- Desarrollo y pruebas. La escritura de un servicio pequeño que utilice otros servicios dependientes requiere un enfoque que no sea escribir una aplicación tradicional monolítica o en capas. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios.
- Falta de gobernanza. El enfoque descentralizado para la generación de microservicios tiene ventajas, pero también puede causar problemas. Puede acabar con tantos lenguajes y marcos de trabajo diferentes que la aplicación puede ser difícil de mantener. Puede resultar útil establecer algunos estándares para todo el proyecto sin restringir excesivamente la flexibilidad de los equipos.
- Integridad de datos. Cada microservicio es responsable de la conservación de sus propios datos. Como consecuencia, la coherencia de los datos puede suponer un problema. Se debe adoptar una coherencia final.
- Control de versiones. Las actualizaciones de un servicio no deben interrumpir servicios que dependen de él. Es posible que varios servicios se actualicen en cualquier momento; por lo tanto, sin un cuidadoso diseño, podrían surgir problemas con la compatibilidad con versiones anteriores o posteriores [11].

La arquitectura de microservicios comenzó a generar expectación tras los conocidos éxitos de Netflix o Amazon. Sin embargo, estas compañías y muchas otras que han conseguido grandes resultados con microservicios tienen una cosa en común: todas son recientes y desarrollaron aplicaciones nuevas. Ninguna de ellas tenía un proyecto en marcha que debiera transformar o reemplazar [5]. Para empresas tradicionales, que existen desde hace años, embarcarse en este modelo cuenta con

la complicación de tener que refactorizar una aplicación monolítica de una envergadura considerable. Sin embargo, en muchas ocasiones son precisamente estas aplicaciones las que más se beneficiarían de una transición a los microservicios [5], debido a su mayor magnitud, número de clientes y constantes actualizaciones.

1.2. Motivación del proyecto

El presente trabajo se centra en el estudio de la transición de una arquitectura monolítica (*Service Oriented Architecture* - SOA) a una de microservicios (*Microservice Architecture* - MSA), la cual se mostrará a través del desarrollo de una aplicación monolítica que posteriormente se transformará en una aplicación basada en microservicios

Surge motivado por mi propia experiencia en el entorno laboral, y como respuesta a la necesidad de unificar la información disponible sobre arquitecturas monolíticas y de microservicios, así como de crear unas directrices generales para la transición de una arquitectura monolítica a una de microservicios.

1.3. Objetivos del proyecto

El objetivo principal de este proyecto es el estudio exhaustivo de las arquitecturas de microservicios, guiado principalmente por el desarrollo de una aplicación, para así obtener una mejor comprensión de las ventajas que pueden aportar a ciertos proyectos, a la par que se muestren sus dificultades y limitaciones.

Con este estudio se pretende además mostrar las diferencias de las arquitecturas basadas en microservicios respecto a las arquitecturas monolíticas. Dichas diferencias se expondrán junto al desarrollo de una aplicación en la que se apreciará la transición de un tipo de arquitectura al otro. Esta transición se logrará mediante el desarrollo de cuatro aplicaciones que se crearán de forma gradual, y con estudiadas diferencias entre ellas. Este proyecto se realiza con la esperanza de que pueda servir como guía para futuros curiosos, estudiantes o desarrolladores.

1.4. Recursos empleados

A continuación se introducirán las herramientas empleadas para la realización de este proyecto.

1.4.1. Eclipse IDE

El entorno de desarrollo integrado (IDE - *Integrated Development Environment*) de Eclipse es una plataforma de desarrollo de *software* de código abierto, diseñada para ser extendida a través de *plug-ins*. Fue concebida desde sus inicios para convertirse en una plataforma de integración de herramientas de desarrollo. No tiene en mente un lenguaje específico, sino que es un IDE genérico, aunque goza de mucha popularidad entre los desarrolladores de Java usando el *plug-in* JDT (*Java Development Tools*), que viene incluido en la distribución estándar de Eclipse.
Proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones [12].

Figura 1.3: Logo de Eclipse IDE

1.4.2. Spring

Spring es un *framework* para crear código de alto rendimiento, liviano y reutilizable. Su finalidad es estandarizar, agilizar y resolver problemas durante el desarrollo.
Spring ofrece suporte a nivel de aplicación, convirtiéndose en un completo modelo para la configuración y programación de aplicaciones empresariales desarrolladas bajo Java. No discrimina en cuanto al despliegue de la plataforma.
Cuenta con plantillas para diversas tecnologías entre las cuales destacan JDBC, Hibernate y JPA. De esta forma no hay necesidad de escribir código extenso, ya que dichas plantillas simplifican el trabajo en cuanto a los pasos básicos a implementar cuando se hace uso de estas tecnologías.
Spring se puede considerar padre de los *framework* Java, ya que da soporte a varios de ellos como Hibernate, Tapestry, EJB o JSF, entre otros [13].

Figura 1.4: Logo de Spring *Framework*

1.4.3. Git

Git es un sistema de control de versiones *open source*, diseñado para manejar cualquier tipo de proyecto de diferentes envergaduras con rapidez y eficiencia [14]. Implementa un sistema de ramas.[en] las que almacenar código, que permite tanto tener registros de distintas versiones a la vez, como permitir que varios miembros de un equipo puedan trabajar simultáneamente sobre el mismo código sin interferencias.

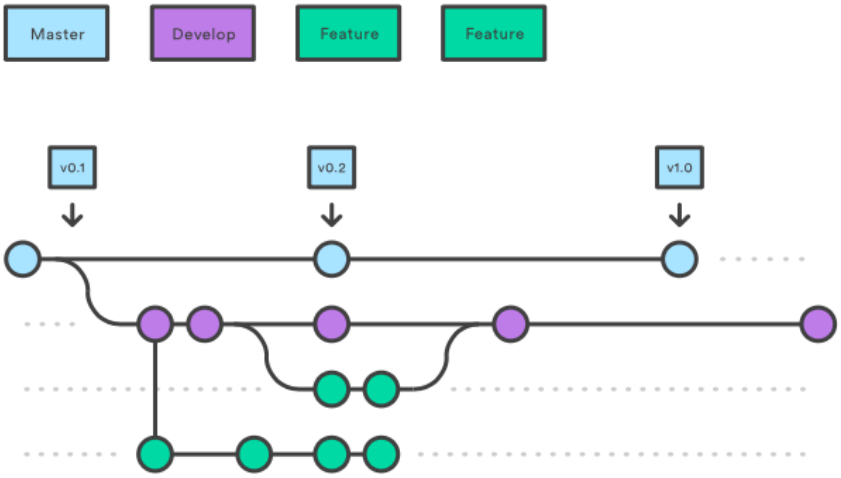


Figura 1.5: Esquema del concepto de ramas de Git

1.4.4. Maven

Apache Maven es una herramienta de administración y comprensión de proyectos. Basada en el concepto POM (*Project Object Model*), Maven se creó en 2001 con el objetivo de simplificar los procesos de *build* (compilar y generar ejecutables a partir del código fuente).
Antes de que Maven proporcionara una interfaz común para hacer *builds* del *software*, cada proyecto solía tener a alguna persona dedicada exclusivamente a configurar ese proceso. Se debía analizar qué partes de código se debían compilar, qué librerías utilizaba el código, dónde incluirlas... Ahora, el proceso *build* de cualquier proyecto Maven, independientemente de sus módulos, consiste simplemente en ejecutar el comando `mvn install` [15].

Figura 1.6: Logo de Apache Maven

1.4.5. MySQL

MySQL es un sistema de gestión de bases de datos relacionales de código abierto con un modelo cliente-servidor. Es un *software* o servicio utilizado para crear y administrar bases de datos basadas en un modelo relacional [16].
Se considera el sistema gestor de bases de datos más popular.

Figura 1.7: Logo de MySQL

1.4.6. JDBC

JDBC (*Java Database Connectivity*) es un API (*Application programming interface*) que describe una librería estándar para acceso a bases de datos desde el lenguaje Java, principalmente orientado a bases de datos relacionales que usan SQL (*Structured Query Language*). JDBC no sólo proporciona un interfaz para acceso a motores de bases de datos, sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos [17].

1.4.7. JPA

Java Persistence API o JPA es la herramienta de Java para implementar un *Framework Object Relational Mapping* (ORM), que permite interactuar con la base de datos por medio de objetos Java. Así, JPA es el encargado de convertir los objetos Java en instrucciones para el sistema de gestión de bases de datos [18].

1.4.8. Hibernate

Hibernate es también una herramienta de mapeo objeto-relacional (ORM) bajo licencia GNU LGPL para Java, que facilita el mapeo de atributos en una base de datos tradicional con el modelo de objetos de un aplicación mediante archivos declarativos o anotaciones en los *beans* de las entidades que permiten establecer estas relaciones. Para realizar esta tarea, permite detallar el modelo de datos, qué relaciones existen y qué forma tienen mediante un documento XML, o mediante anotaciones donde corresponde un atributo de una clase, con una columna de una tabla[19].

1.4.9. H2

H2 es una base de datos ligera *open.source* de Java. Puede embeberse en aplicaciones Java, o ejecutarse en modo cliente-servidor. Puede configurarse de forma que los datos no persistan en el disco [20].

1.4.10. Postman

Postman surgió como una extensión para el navegador Google Chrome. A día de hoy dispone de aplicaciones nativas para distintos sistemas operativos.
Se compone de diferentes herramientas que permiten realizar diversas tareas dentro del mundo API REST, como son la creación de peticiones a APIs internas o de terceros, la elaboración de *tests* para validar el comportamiento de APIs y la

posibilidad de crear entornos de trabajo diferentes con variables locales y globales. El interés fundamental de Postman es que su uso como una herramienta para hacer peticiones a APIs y generar colecciones de peticiones que nos permitan probarlas de una manera rápida y sencilla. Las colecciones son carpetas donde se almacenan las peticiones y que permiten ser estructuradas por recursos o módulos. Además, estas colecciones pueden ser compartidas [21].

Figura 1.8: Logo de Postman

Capítulo 2

Arquitectura del proyecto

Una vez se ha introducido el contexto del proyecto y se han puesto de manifiesto los motivos que llevan a realizar este estudio, además de los objetivos que se pretenden lograr con él, se puede dar paso a la explicación de la arquitectura del sistema. Este capítulo pretende servir como guía inicial de la arquitectura general, puesto que se continuarán detallando las cuatro diferentes arquitecturas en sus respectivos capítulos.

2.1. API REST

Al ser todas ellas aplicaciones REST, mantienen ciertas características en común a pesar de tener arquitecturas diferentes, puesto que dos de ellas son monolíticas y dos de microservicios. El siguiente esquema ilustra las partes comunes a todas las aplicaciones. En distintos tonos de azul se observan los *packages* y, en blanco, los archivos de código Java.

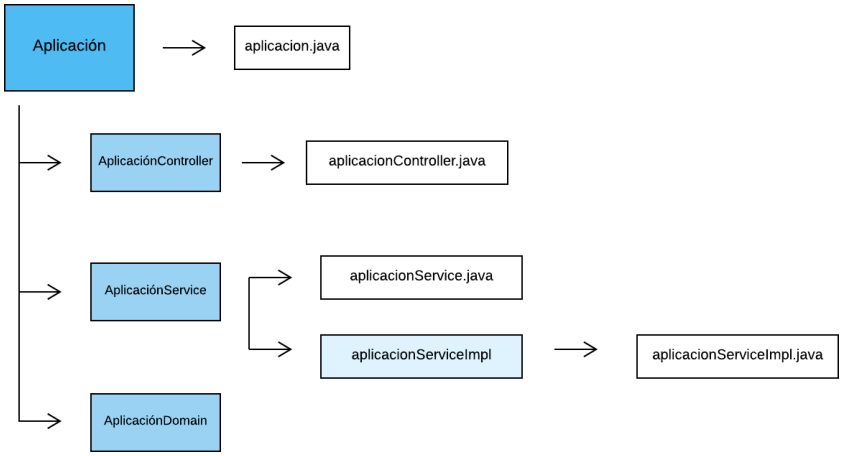


Figura 2.1: Arquitectura común a todas las aplicaciones

El archivo situado en la parte superior de la figura contiene el método *main*, que inicializa la aplicación. En un nivel inferior, se encuentra el *controller*, que es quien recibe las peticiones del cliente. Estas, como ya se ha explicado, pueden ser de tipo GET, POST, etcétera. Una vez recibidas, el *controller* llama al método correspondiente del *service*, que en la imagen se muestra en el nivel inmediatamente inferior, para procesar cada petición.

Dado que el *service* es una interfaz, requiere de una implementación, que aparece reflejada en el esquema a la derecha del *service*.

Finalmente, en la parte inferior de la imagen se observa el dominio. Su contenido será similar en todas las aplicaciones, pero no idéntico. Se especificarán las diferencias de cada *domain* en sus respectivos capítulos.

En cuanto al diseño del servicio REST, este capítulo presentará una versión general que no se corresponde con todas las aplicaciones. Sin embargo, en los capítulos correspondientes se detallarán las diferencias respecto al modelo que aquí se introduce.

En la siguiente figura se muestra el diseño del servicio *store*, que será el principal y común a todas las aplicaciones. Posteriormente, la última de ellas verá la adición de otro servicio que no será detallado en este capítulo.

La forma de acceder a los recursos que aparecen en la imagen será anexionar los términos indicados en la imagen a la URI `http://localhost:8080/store`.

Así, en la parte superior del esquema se encuentra el método `getCartItems()`. La acción que implementa sobre los datos es GET, y permite obtener el listado

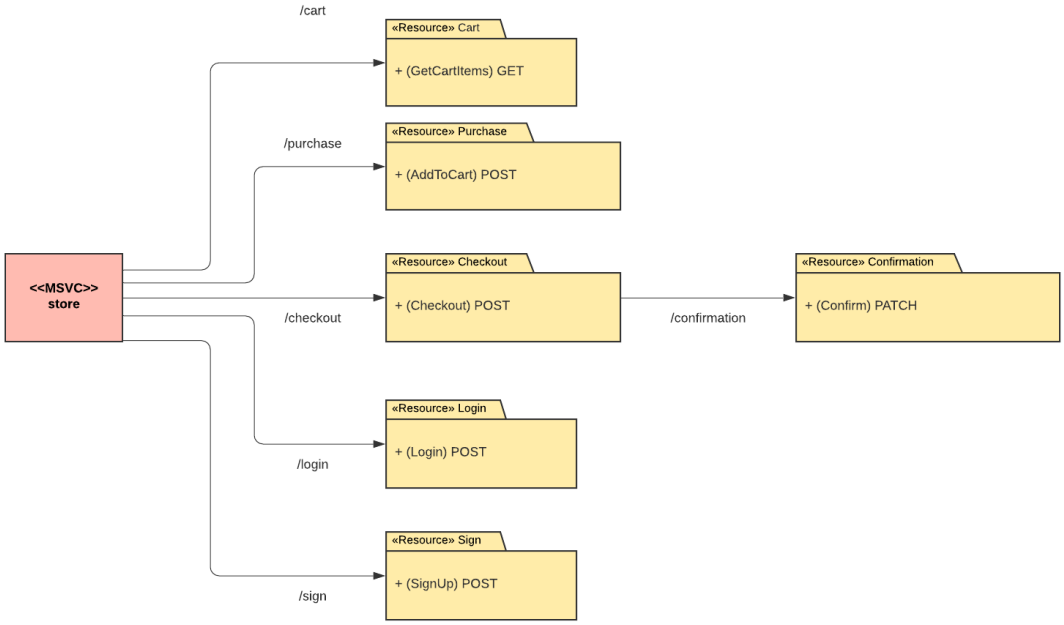


Figura 2.2: Servicio *store*

de artículos que han sido añadidos al carro de compra. Se accede a este recurso mediante la URI `http://localhost:8080/store/cart`.

El siguiente método en la figura es `addToCart()`. La acción que este método implementa sobre los datos es POST. Esto implica que, a diferencia del método anterior, cuenta con la posibilidad de enviar un *body* que será recibido por el *controller* de la aplicación, como se explicó en la figura 2.1. Este método recibe la referencia del artículo que se quiere añadir al carro de compra e introduce dicho artículo en una estructura de datos temporal (el carro de compra) hasta que se formaliza el pedido. Se accede a este recurso por medio de la URI `http://localhost:8080/store/purchase`, y su *body* es un Integer correspondiente a la referencia del producto.

El método localizado en el centro de la figura es `checkout()`, cuya acción sobre los datos es POST. Sin embargo, y, a diferencia del anterior, el *body* de este método se enviará vacío. Este método no es más que el primer paso para realizar el pedido, y muestra los artículos contenidos en el carro. Se accede a él a través de la URI `http://localhost:8080/store/checkout`.

A continuación se encuentra el método que oficializa la compra de los artículos

que el método `checkout()` acaba de mostrar. Este es `confirm()`, que realiza un PATCH. Se crea un pedido con los productos que hasta ahora se almacenaban en esa estructura de datos temporal y se actualiza su *stock*. Se accede a esta funcionalidad mediante la URI `http://localhost:8080/store/checkout/confirmation`.

Los dos últimos métodos son `login()` y `signUp()`. Ambos realizan un POST y requieren de un *body*. En el caso de `login()`, dicho cuerpo contiene el *email* y contraseña del usuario. Para `signUp()`, se requieren además el nombre, apellido y dirección del usuario. Con esta información, se insertará un nuevo usuario en la base de datos. `http://localhost:8080/store/login` es la ruta para acceder al primero y a través de `http://localhost:8080/store/sign` se accede al segundo.

2.2. Base de datos

Finalmente, se introduce la base de datos del sistema. Esta también mostrará diferencias dependiendo de la aplicación, que serán especificadas en sus respectivos capítulos.

En la siguiente figura aparecen representadas las tablas `product`, `orders`, `user` y `role`. La discordancia en el nombre de la tabla `orders` se debe a que el término *order* es una palabra reservada en SQL y, por tanto, no se permite denominar así a una tabla.

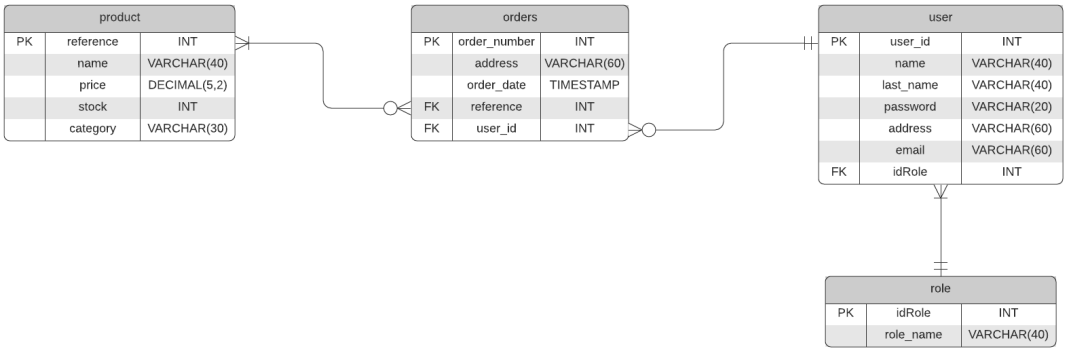


Figura 2.3: Modelo Entidad Relación del proyecto

La tabla `product` está compuesta por las siguientes columnas:

1. `reference`, que es la clave primaria de la tabla y representa la referencia del producto.
2. `name`, que representa el nombre del producto.
3. `price`, correspondiente al precio del producto.
4. `stock`, que indica la cantidad del producto que el sistema tiene disponible.
5. `category`, que señala el tipo de producto.

La tabla `orders` contiene las siguientes columnas:

1. `order_number`, que es la clave primaria de la tabla y representa el número de pedido.
2. `address`, que indica la dirección en la que debe entregarse el pedido.
3. `order_date`, que muestra la fecha en la que se realizó el pedido.
4. `reference`, que es clave foránea de la tabla `product` y permite identificar qué artículos se compraron en el pedido en cuestión.
5. `user_id`, que es clave foránea de la tabla `user` e indica qué usuario realizó el pedido.

La tabla `user` contiene las siguientes columnas:

1. `user_id`, que es la clave primaria de la tabla e identifica al usuario.
2. `name`, que se corresponde con el primer nombre del usuario.
3. `last_name`, que representa el apellido del usuario.
4. `password`, que es la contraseña que permite al usuario acceder a su cuenta.
5. `address`, que indica la dirección en la que reside el cliente.
6. `email`, con el que el usuario accede a su cuenta.
7. `idRole`, que es clave foránea de la tabla `role` e indica el tipo de usuario.

La cardinalidad de las relaciones entre las tablas es la siguiente:

- Las tablas `product` y `orders` tienen relación uno/muchos a ninguno/muchos.
- Las tablas `user` y `orders` tienen relación uno a ninguno/muchos.
- Las tablas `user` y `role` tienen relación uno/muchos a uno.

Capítulo 3

Aplicación monolítica inicial

En este capítulo se describe con detalle la primera aplicación de las cuatro que conforman el proyecto. Además, se ilustrará la creación y configuración inicial de una aplicación Spring Boot y de un repositorio Git. Dicho proceso también se llevó a cabo en el resto de aplicaciones, por lo que no será necesario exponerlo de nuevo en sus respectivos capítulos. También es importante indicar que, en adelante, habrá alusiones a cada una de las aplicaciones como "versión". Esta, en concreto, será la versión uno, puesto que así se ha denominado durante su programación.

3.1. Primeros pasos

3.1.1. Aplicación Spring Boot

Para generar el esqueleto de la aplicación Spring Boot, se dispone de la web Spring Initializr, disponible en `http://start.spring.io/` y cuya interfaz se muestra en la figura 3.1. En ella, se puede elegir una variedad de posibles configuraciones. En este caso se selecciona que el proyecto sea Maven, su lenguaje Java 8 y se da nombre a los distintos campos. Al pulsar *Generate*, se descarga un zip con el esqueleto resultante. A pesar de no contener aún ninguna funcionalidad correspondiente al proyecto, este esqueleto ya puede ser ejecutado, puesto que contiene un método `main()`:

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

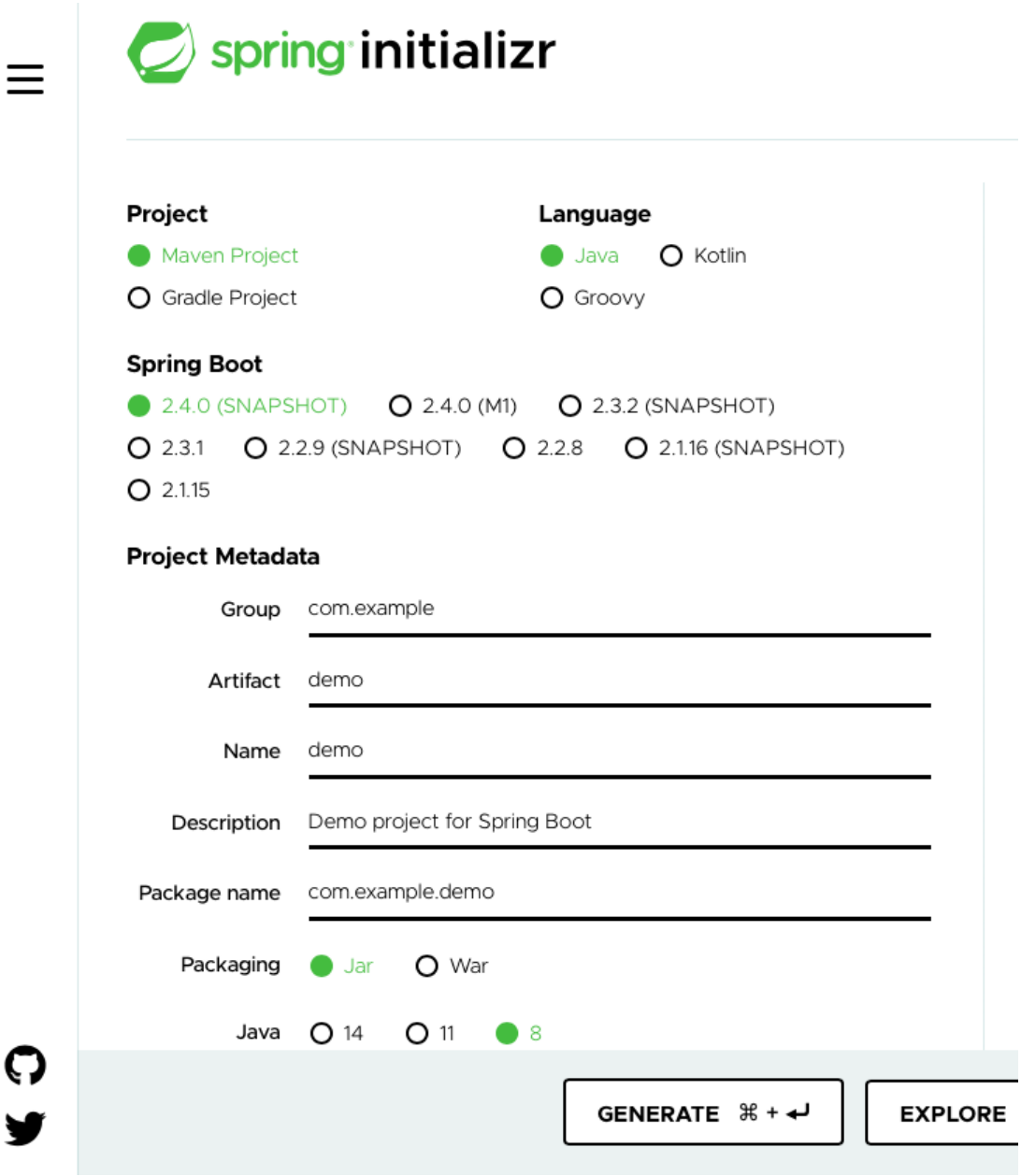


Figura 3.1: Interfaz de Spring Initializr

El siguiente paso es importar el proyecto generado por Spring Boot a Eclipse. Para ello, se selecciona la opción *Import... Existing Maven Projects*. Una vez finalizado esto, el proyecto generado por Spring Boot está listo para ser modificado y ejecutado en Eclipse.

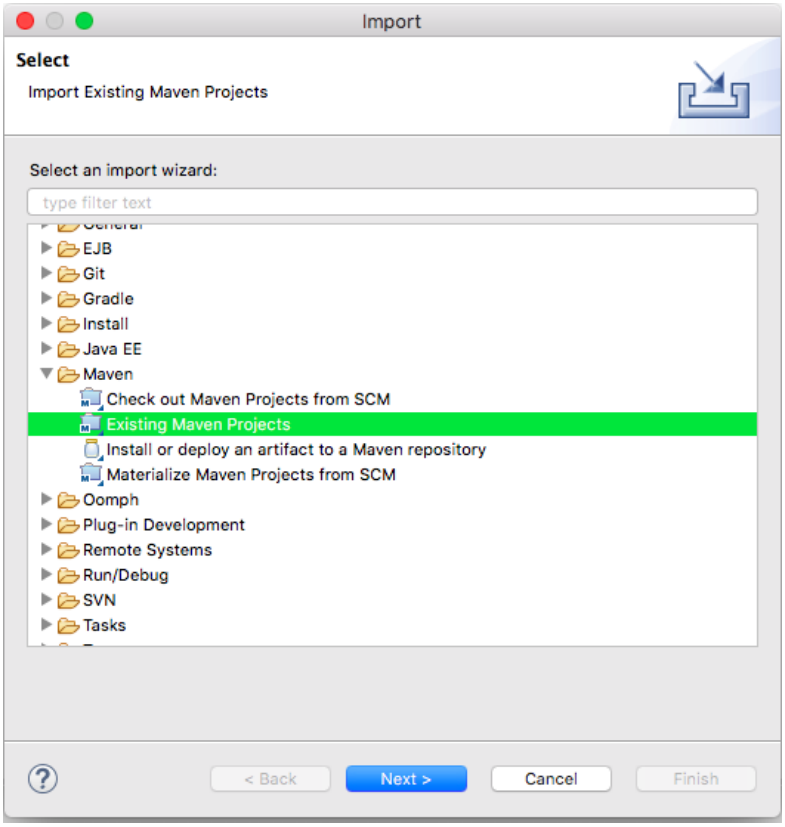


Figura 3.2: Menú para importar el proyecto generado por Spring Initializr

3.1.2. Repositorio de Git

Tras dar de alta una cuenta en Git, se pueden crear repositorios en los que almacenar código. En este caso, se crea un repositorio para cada versión. Se encuentran en `https://github.com/anahuertat`.

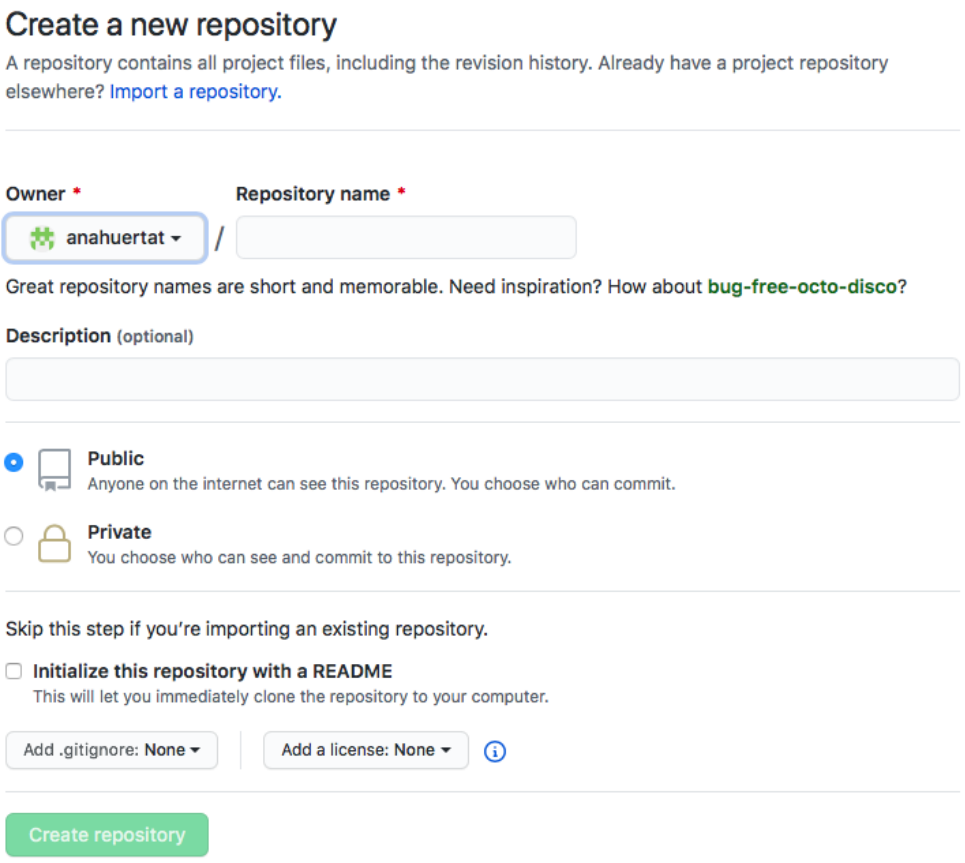


Figura 3.3: Interfaz para la creación de repositorios Git

En la figura 3.3 se muestra el menú que permite crear un nuevo repositorio. Una vez creado, se debe clonar este repositorio en el equipo para así tener una versión local que poder modificar y subir al repositorio. Esto se puede hacer desde la línea de comandos o, de forma más cómoda, desde la aplicación de escritorio de Git (figura 3.4), que se conecta a la cuenta, detecta modificaciones en el código almacenado en local de cada uno de los repositorios y permite realizar `commit` con un clic.

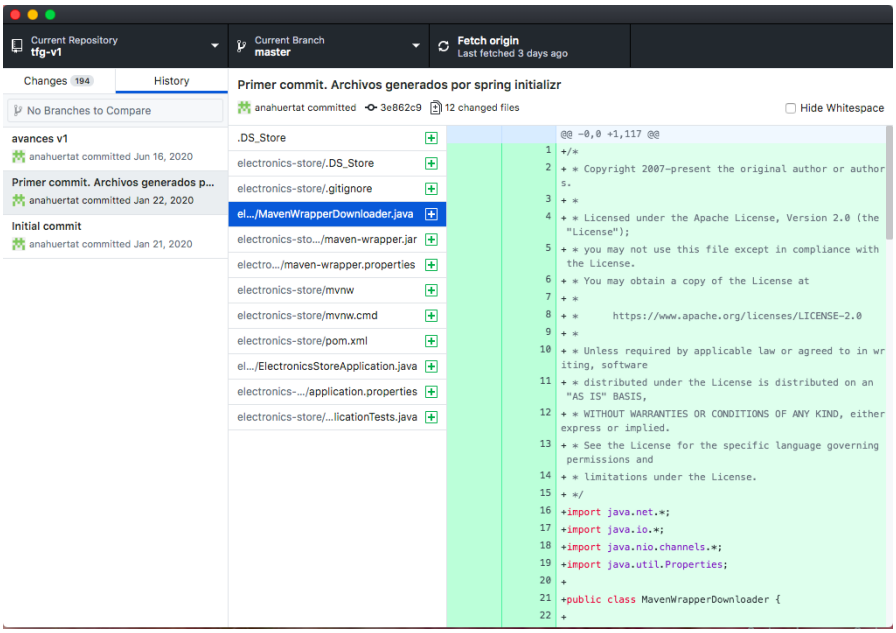


Figura 3.4: Aplicación *GitHub Desktop* para facilitar el control de versiones

3.2. Desarrollo

El desarrollo de esta aplicación comienza con la definición del dominio. Este se compondrá de las clases *Cart*, *Category*, *Component* y *Components*.

Cart representa el carro de la compra. Es una lista de productos que el usuario añade hasta que decide oficializar la compra.

Category es una enumeración de las distintas categorías en las que pueden clasificar los diferentes productos disponibles.

Component representa cada uno de los productos disponibles para ser comprados. Sus atributos son *Category*, *name*, *reference*, *price* y *stock*.

Components se crea para suplir a la base de datos, que no forma parte de esta versión. Consiste en una lista de productos inicializados de forma manual, que serán los que se encuentran a la venta. Cuando se realizan compras, disminuye el registro de *stock* en dicha lista.

El siguiente objeto de estudio de esta sección es el *service*. Se necesitan métodos para añadir productos al carro, listar los productos que se encuentran en el carro y formalizar la compra. Todos ellos se definen en la interfaz `ElectronicsStoreService.java` y se implementan en `ElectronicsStoreServiceImpl.java`.

Para el método `addToCart()`, se recibe una referencia, que se añade a una instancia de `Cart()`. Es decir, *Cart* almacena inicialmente una lista de referencias,

no de productos.

Es al listar los productos cuando esa lista de referencias se recorre para obtener la información restante de los productos correspondientes.

Por último, al finalizar la compra se modifican las existencias disponibles de cada producto en la lista inicializada manualmente que cumple las funciones de base de datos de forma provisional.

Finalmente, se necesita de un *controller* que reciba las peticiones del cliente y las procese de la manera que corresponda.

En este caso, y de acuerdo a los métodos discutidos anteriormente, se determina que para listar los productos del carro de compra se empleará un GET. A continuación, el *controller* llamará al método descrito en esta sección.

Asimismo, para añadir un artículo al carro de compra se utilizará un POST y se llamará al método `addToCart()` ya descrito.

Finalmente, para completar la compra será necesario un PATCH seguido de la llamada al método que modifica la lista de productos.

Esta aplicación tiene una estructura más pulida y menos propensa a errores gracias a su definición como aplicación REST, así como al esqueleto proporcionado por Spring Boot.

En un monolito clásico, el método `main()` sería el lugar en que se llevaría a cabo toda la funcionalidad que en esta estructura se ha repartido en los *services* y *controller*. Este hecho es algo a tener en cuenta cuando se intentan comprender las diferencias entre la arquitectura monolítica y la de microservicios.

También es importante mencionar que las diferencias que se observarán a lo largo del proyecto son tan solo una pequeña muestra, ya que estas se acentúan con aplicaciones de mayor envergadura que escapan al alcance de este proyecto.

Capítulo 4

Aplicación monolítica final

En este capítulo se describe detalladamente el desarrollo de la segunda aplicación. Esta versión parte del desarrollo explicado en el capítulo anterior, completándolo con una base de datos. Este capítulo contendrá, además, una minuciosa explicación del proceso de diseño e implementación de la capa de datos del proyecto.

4.1. Capa de Datos

La aplicación del capítulo 3 no podría ser una aplicación monolítica completa y realista sin una capa de datos en la que almacenar los productos disponibles y los pedidos realizados por el cliente.

Para su diseño, es necesario tener un gran entendimiento de las clases que conforman el dominio de la aplicación, de las relaciones entre ellas que mejor modelan la realidad y de las posibilidades que el diseño de bases de datos nos ofrece.

Puesto que hay diferentes clases en el dominio respecto a la versión del capítulo 3, el primer paso será introducirlas para más tarde comprender mejor los requerimientos de la base de datos.

Cart, al igual que en la versión anterior, representa el carro de la compra, instanciado como una lista en la que almacenar los artículos temporalmente hasta que la compra se finaliza.

Category, como en el capítulo 3, es una enumeración de las posibles categorías de artículos.

Component es cada uno de los productos disponibles en la tienda. Sus atributos son:

- *Category*, que se corresponde con la enumeración ya descrita,
- *name*, el nombre del producto,
- *reference*, el identificador único del producto,
- *price*, el precio del producto,
- *stock*, las unidades disponibles del producto.

Order es una nueva clase. Representa un pedido formalizado. Sus atributos son:

- *order_number*, identificador único del pedido,
- *address*, dirección de entrega del pedido,
- *order_date*, fecha de compra,
- *user*, usuario que realizó el pedido.

User también es una nueva clase. Representa un usuario registrado en la página. Sus atributos son:

- *user_id*, identificador único del usuario,
- *name*, primer nombre del usuario,
- *last_name*, apellido del usuario,
- *address*, dirección del cliente,
- *email*,
- *password*, contraseña de acceso del usuario,
- *orders*, historial de pedidos del usuario.

Cart no necesitará persistir en la base de datos, puesto que es un almacenamiento temporal. En cuanto a las demás, se necesitará al menos una tabla *order*, una *user* y otra *component*.

Asimismo, el sistema debe contar con una distinción de roles de los usuarios para poder administrar los distintos permisos de lectura y escritura de la aplicación. Es decir, algunos usuarios podrán añadir o eliminar artículos de las existencias, o

modificar sus precios, por ejemplo, mientras que otros solo podrán realizar compras.

Añadiendo la tabla *role*, se dispone pues de 4 tablas. A continuación se estudiarán las relaciones entre ellas.

1. Un producto puede estar presente en numerosos pedidos, o en ninguno. La cardinalidad es, por tanto, ninguno o muchos.
2. Un pedido debe incluir como mínimo un producto. También puede contener varios artículos. Por ello, la cardinalidad es uno o muchos.
3. Un usuario puede haber realizado muchos pedidos, o ninguno. Según esto, su cardinalidad es ninguno o muchos.
4. Un pedido puede estar asociado a un único cliente. La cardinalidad es uno.
5. Un usuario tendrá un solo rol. La cardinalidad es uno.
6. Un rol puede estar asociado desde uno hasta numerosos clientes. La cardinalidad será, por tanto, uno o muchos.

Según las especificaciones anteriores, el diseño de base de datos resultante es el que se muestra en la figura 4.1. Tanto *reference* como *user_id* se convierten en claves foráneas de la tabla *order*, puesto que desde dicha tabla se debe poder asociar el usuario y los artículos que componen el pedido.

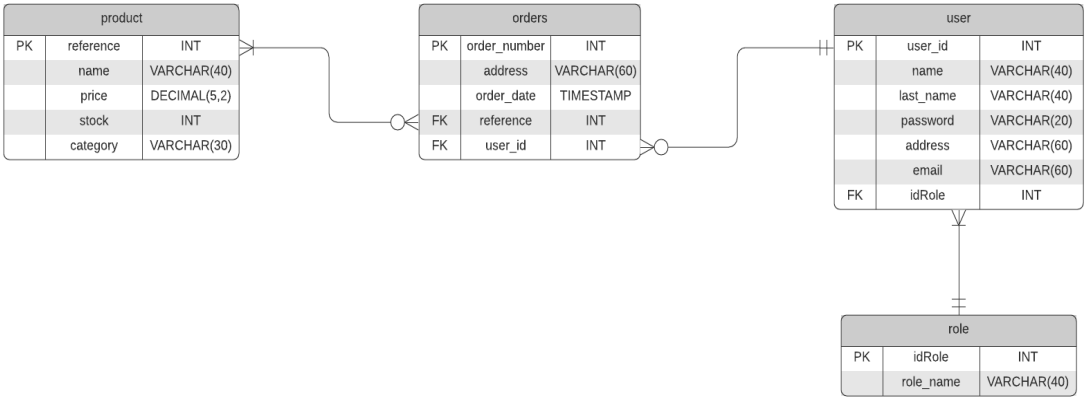


Figura 4.1: Diseño de la base de datos

El siguiente paso es crear estas tablas. Esto se hará desde la línea de comandos de MySQL, después de haberlo instalado. En la figura 4.2 se muestra cómo acceder a dicha consola.

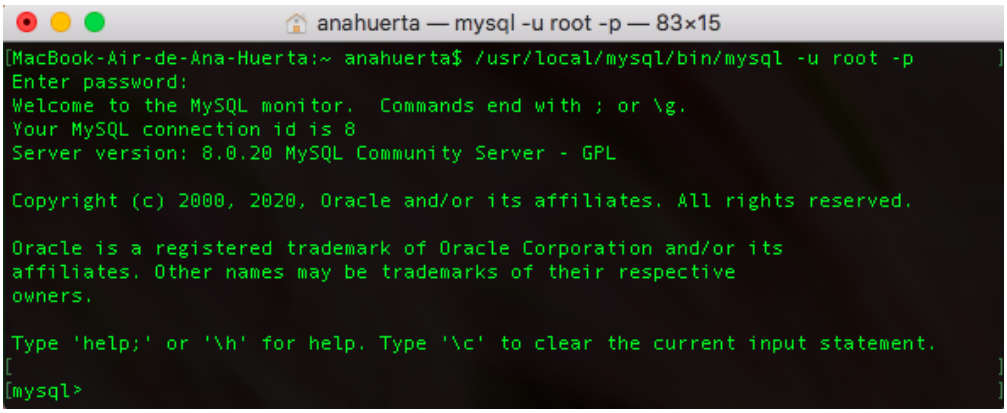


Figura 4.2: Acceso al terminal de MySQL

Una vez se ha accedido, debe crearse una base de datos en la que almacenar las tablas descritas. Se denominará a dicha base de datos `tfg`. Ahora sí, se crean las tablas mediante el comando SQL `create table`. La figura 4.3 muestra el proceso de creación de la tabla *orders*.

Tras crearlas, se deberán añadir los *constraints* correspondientes, como por ejemplo determinar qué columnas no pueden contener un valor `null`, crear las claves foráneas, especificar valores por defecto e introducir la condición de que los valores de las columnas *reference*, *order_number* y *user_id* incrementen en uno de forma automática con cada nuevo registro. Estos detalles aparecen en la figura 4.4.

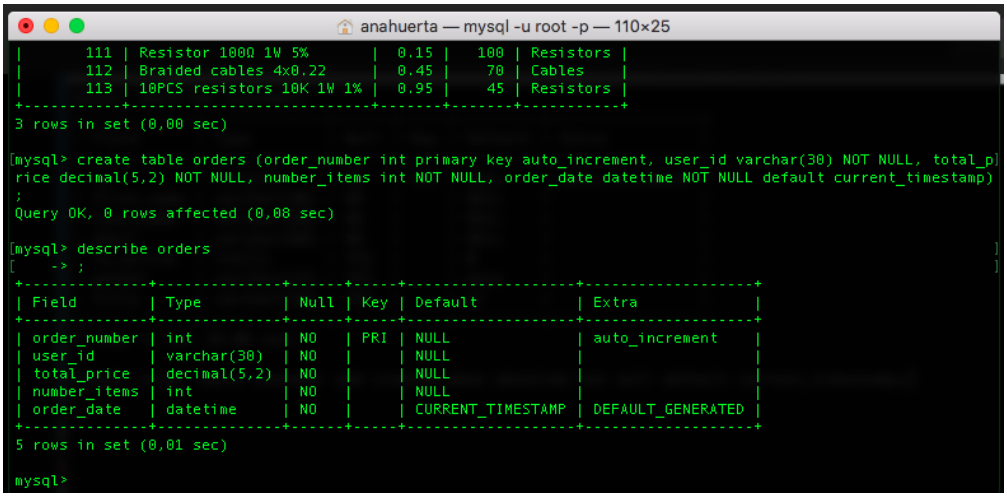
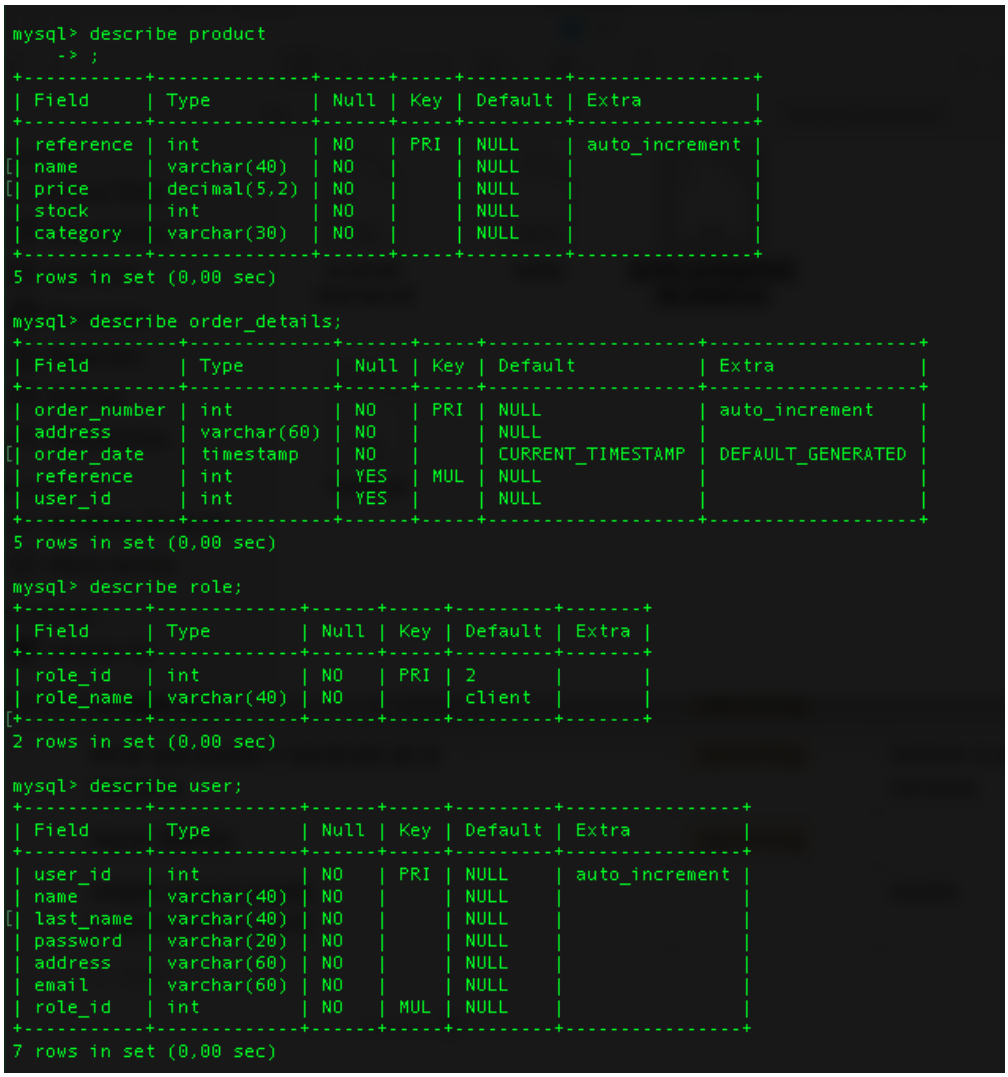


Figura 4.3: Creación de una de las tablas del sistema.



```
mysql> describe product
    -> ;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| reference | int         | NO   | PRI | NULL    | auto_increment |
| name      | varchar(40) | NO   |     | NULL    |                |
| price     | decimal(5,2)| NO   |     | NULL    |                |
| stock     | int         | NO   |     | NULL    |                |
| category  | varchar(30) | NO   |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
5 rows in set (0,00 sec)

mysql> describe order_details;
+--------------+-------------+------+-----+-------------------+-------------------+
| Field        | Type        | Null | Key | Default           | Extra             |
+--------------+-------------+------+-----+-------------------+-------------------+
| order_number | int         | NO   | PRI | NULL              | auto_increment    |
| address      | varchar(60) | NO   |     | NULL              |                   |
| order_date   | timestamp   | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
| reference    | int         | YES  | MUL | NULL              |                   |
| user_id      | int         | YES  |     | NULL              |                   |
+--------------+-------------+------+-----+-------------------+-------------------+
5 rows in set (0,00 sec)

mysql> describe role;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| role_id   | int         | NO   | PRI | 2       |       |
| role_name | varchar(40) | NO   |     | client  |       |
+-----------+-------------+------+-----+---------+-------+
2 rows in set (0,00 sec)

mysql> describe user;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| user_id   | int         | NO   | PRI | NULL    | auto_increment |
| name      | varchar(40) | NO   |     | NULL    |                |
| last_name | varchar(40) | NO   |     | NULL    |                |
| password  | varchar(20) | NO   |     | NULL    |                |
| address   | varchar(60) | NO   |     | NULL    |                |
| email     | varchar(60) | NO   |     | NULL    |                |
| role_id   | int         | NO   | MUL | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
7 rows in set (0,00 sec)
```

Figura 4.4: Resultado final del diseño y creación de las tablas de datos.

Por último, se deben introducir registros en cada una de las tablas, haciendo uso del comando SQL *insert into table*. En la figura 4.5 se muestra un ejemplo de esta tarea.

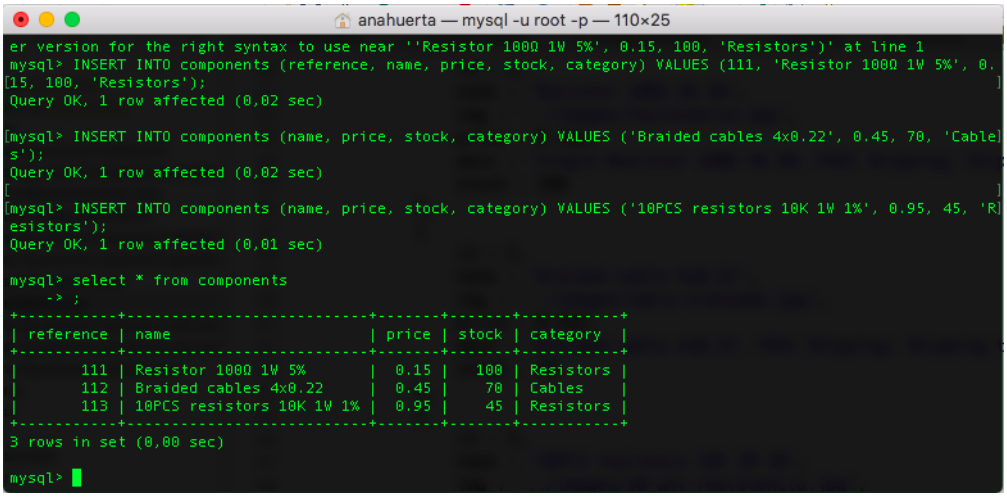


Figura 4.5: Ejemplo de inserción en una de las tablas.

4.2. Desarrollo

Tras la introducción y desarrollo de la capa de datos, *domain* incluido, en el apartado anterior, se procede a la incorporación de las nuevas tablas de datos a la aplicación Java. JDBC o *Java DataBase Connectivity* permite establecer la conexión con la base de datos y operar sobre ella. Para llevar a cabo esta tarea, se crea la clase `StoreDataBase.java` dentro del nuevo paquete *persistence*.

En dicha clase se define cada uno de los métodos que requerirán de acceso a la base de datos. Todos ellos deberán implementar una serie de pasos comunes para asegurar una conexión exitosa, y siempre incluidos en un bloque `try - catch`. Estos son:

1. Crear la conexión con la base de datos deseada.

```
//1. create connection
Class.forName("com.mysql.cj.jdbc.Driver");
connection = DriverManager.
 getConnection("jdbc:mysql://localhost/tfg", USER,
 PASSWORD);
```

2. Preparar la sentencia SQL.

```
//2. Prepare statement
String query = "SELECT * FROM product WHERE
 reference=?";
preparedStatement = connection.prepareStatement(query);
```

3. En el caso de una *query* dinámica, establecer los parámetros en la sentencia.

```
//3. Set parameters in statement
preparedStatement.setInt(1, reference);
```

4. Ejecutar la consulta.

```
//4. Execute query
resultSet = preparedStatement.executeQuery();
```

5. Procesar el resultado de la consulta.

```
//5. Extract data from resultset
if(resultSet.next()) {
 component = new Component(
 resultSet.getString("category"),
 resultSet.getString("name"),
 resultSet.getDouble("price"),
 reference,
 resultSet.getInt("stock"));
}
```

6. Limpiar el entorno y cerrar la conexión.

```
//6. Clean-up environment
resultSet.close();
preparedStatement.close();
connection.close();
```

El código mostrado como ejemplo se corresponde con el método que obtiene un producto conocida su referencia. Se implementan métodos similares para obtener y para actualizar el *stock* de un producto dada su referencia, para permitir al cliente hacer *login* y acceder a su cuenta y para crear un pedido.

La implementación del *controller* es muy similar a la del capítulo 3, excepto por la adición del método para realizar el *login*. Este es un POST, que recibe un *email* y una contraseña.

En cuanto a los nuevos *service* y *serviceImpl*, la principal diferencia será que ya no realizan cambios sobre `Components` sino que realizan llamadas los métodos de `StoreDataBase`, que se conectan a la base de datos y obtienen información o la modifican.

Capítulo 5

Aplicación de microservicios inicial

En el presente capítulo se detalla el desarrollo de la tercera versión, que supondrá el cambio a una arquitectura de microservicios. A pesar del cambio, las tareas realizadas por esta aplicación serán las mismas que las del capítulo anterior, más la adición de funcionalidad que permita a los usuarios darse de alta en el sistema.

5.1. Cambios en la Capa de Datos

En el capítulo 4 se estudió el proceso de diseño y creación de la base de datos del sistema, y la posterior conexión desde Java empleando JDBC.

Sin embargo, en este capítulo se sacará partido de las facilidades que nos brindan Java Persistence API y Spring Data.

5.1.1. Java Persistence API

En los capítulos 3 y 4 se crearon las clases Java que se consideraron apropiadas y, después, fue necesario estudiar cómo traducirlas a distintas tablas de datos y cuáles debían ser las relaciones entre ellas.

En la aplicación que se describe en este capítulo, se tomarán las clases Java creadas en versiones anteriores y, gracias al uso de JPA (Java Persistence API) y sus anotaciones, estas serán traducidas a una base de datos de manera automática.

En este caso, la base de datos que se empleará no será MySQL como en la versión dos, sino H2. Para incorporarla al proyecto, debemos declararla como dependencia en el archivo `pom.xml`.También se debe añadir la de JPA:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

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

A continuación se presentan los cambios que Java Persistence API introduce en el dominio.

El primer paso será incluir la anotación `@Entity` antes de la declaración de la clase. Esto le indica a JPA que debe crear una tabla bajo este nombre. A continuación habrá que añadir sobre cada atributo la anotación `@Column(name = "nombreDeColumna")`. De esta forma, JPA sabrá qué columnas introducir en dicha `Entity`.

Además de declarar las columnas, se debe identificar la clave primaria de la tabla. Para ello se emplea la anotación `@Id`. En este caso se añadirá a la clave primaria la anotación `@GeneratedValue`, que permitirá que dicho atributo se incremente automáticamente con cada registro. Dicho comportamiento lo exhibían también las claves primarias de las tablas MySQL.

El siguiente paso será la importante labor de relacionar unas entidades con otras. Para ello se tendrá en cuenta la cardinalidad estudiada anteriormente, y se ligarán unas columnas con otras gracias a anotaciones como `@OneToOne`, `@OneToMany`, `@ManyToMany`, etcétera.

De acuerdo a las relaciones entre tablas ya explicadas, las anotaciones que se deberán emplear son las siguientes:

- En la clase `Orders.java` se incluye una relación `@ManyToOne` con la tabla `User`. Además, se debe especificar qué columna de dicha tabla establece la unión, o en otras palabras, cuál será la clave foránea de esta tabla.

```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name = "user_id")
public User user;
```

- Ya que la clase `Orders.java` contenía dos claves foráneas, se debe añadir además una relación `@ManyToMany` con la tabla `Component`. Este tipo de

relación crea automáticamente una tabla intermedia, que en este caso se denominará `order_has_component`. Dicha tabla contiene tan solo dos columnas: la clave primaria de cada una de las dos tablas que se unen. En este caso, serán `order_number` y `reference`. En cuanto a la clase Java, esta unión da lugar a una lista de componentes que forman parte de este pedido. Puesto que las dos partes de la relación se han tenido en cuenta al crear la nueva tabla intermedia, no será necesario añadir una anotación en la clase `Component.java`.

```
@ManyToMany(cascade = CascadeType.PERSIST)
@JoinTable(
  name = "order_has_component",
  joinColumns = @JoinColumn(name= "order_number"),
  inverseJoinColumns = @JoinColumn(name = "reference"))
public Set<Component> componentsInThisOrder;
```

- En último lugar, habrá que incluir en la clase `User.java` una relacion `@OneToMany` que complemente la que se estableció en la clase `Orders`. Esta da lugar a una lista de los pedidos realizados por el usuario.

```
@OneToMany(cascade = CascadeType.ALL)
public List<Orders> orders;
```

Por último, JPA requiere que el constructor se deje vacío. Las tablas creadas de esta manera serán accesibles únicamente cuando la aplicación se haya puesto en marcha, a través de `http://localhost:8080/h2-console`. Se solicitarán credenciales de acceso. En este caso se han mantenido los valores por defecto, es decir, usuario "saz contraseña vacía, tal y como se observa en la figura 5.1.

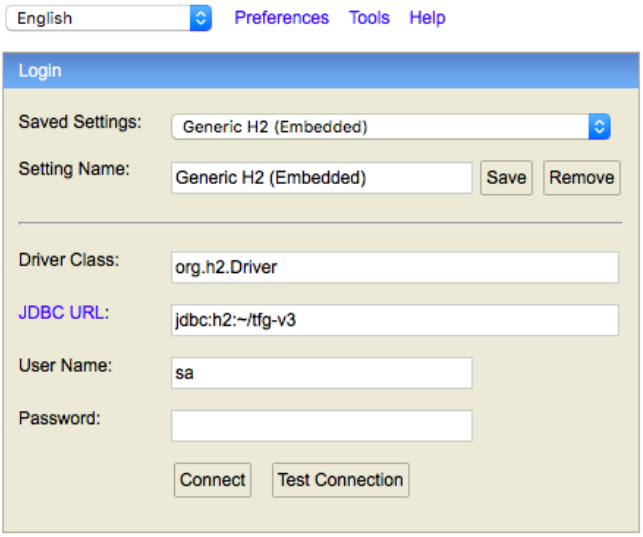


Figura 5.1: Acceso a la consola de H2.

Una vez se ha accedido a la consola, se dispone del listado de tablas a la izquierda, de una pequeña área de texto en la que escribir y ejecutar comandos a la derecha y, en la parte inferior, de sugerencias de comandos e información de ayuda.

En la figura 5.2 se observa que en la lista de tablas se encuentran las mencionadas anteriormente, además de las nuevas que surgen al enlazar tablas. Al acceder por primera vez a la consola, se añaden los valores por defecto que se determinaron en el capítulo anterior y se insertan algunos valores en cada tabla.

La configuración por defecto de H2 implica que las tablas no persisten una vez se detiene la aplicación. Ya que esto no encaja con el comportamiento deseado, se edita el archivo `application.properties`:

```
# Access to the H2 database web console
spring.h2.console.enabled=true
# Generates the database only if it's not there yet
spring.jpa.hibernate.ddl-auto=update
# creates the database in a file
spring.datasource.url=jdbc:h2:file:~/tfg-v3;
DB_CLOSE_ON_EXIT=FALSE;
# show sql in console
spring.jpa.properties.hibernate.show_sql=true
```

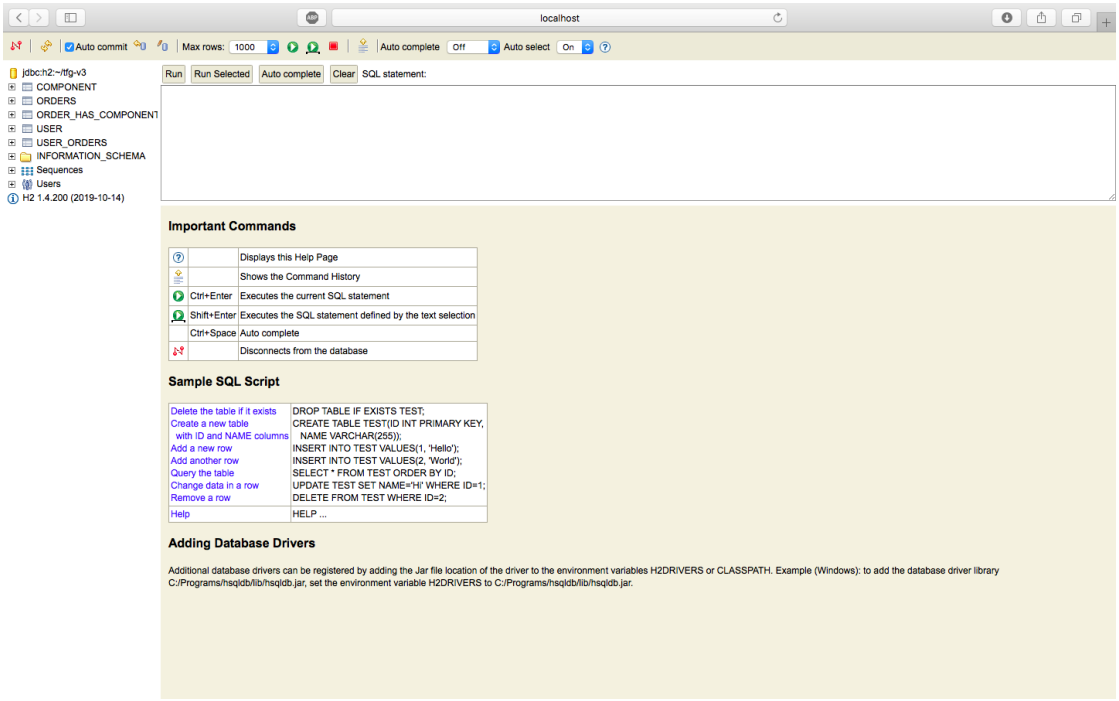


Figura 5.2: Apariencia de la consola de H2.

5.1.2. Spring Data

En el capítulo 4 se introdujo la conexión a una base de datos a través de JDBC. Se trataba de una única clase Java en la que cada método debía implementar manualmente su propia conexión y posterior desconexión. Era, por tanto, un proceso pesado y redundante, y todos los métodos eran de una longitud considerable.

En esta nueva versión, sin embargo, esas desventajas se eliminan al utilizar Spring Data y sus repositorios.

Las interfaces *JpaRepository* y *CrudRepository* implementan una serie de métodos, tales como `findAll()`, `saveAll()`, `deleteAll()` o `findAllById()`. Estos pueden ser útiles. Sin embargo, el gran potencial de ambos repositorios no recae en estos métodos, sino en la posibilidad de adaptarlos al propio código tan solo cambiando el nombre del método. Este tipo de métodos se denominan *Derived Query Methods*, o métodos de consulta derivados.

Para hacer uso de ellos, se debe entender que el nombre del método tiene dos partes diferenciadas por la palabra *By*: en `findByName()`, *find* es la introducción y *name* es el criterio. Además, se pueden añadir delimitadores del resultado tales como *Distinct, First*, o *Top*, por ejemplo, `findTop3ByAge()` permitiría obtener

solo tres resultados.

Otra opción es la de realizar consultas en las que se desea obtener un único resultado, es decir, que un atributo del resultado coincida exactamente con un parámetro que se envía con la consulta. Para lograr, por ejemplo, encontrar un único artículo por nombre, se emplearía `findByName(String name)`. Esto es equivalente a `findByNameIs(String name)`, que se puede usar indistintamente para mayor claridad. Otra posibilidad es desear la no coincidencia con un parámetro, y en tal caso se definiría como `findByNameIsNot(String name)`.

Asimismo, se pueden adaptar más estos métodos con la adición de los términos *And* y *Or* si se desea hacer una consulta con múltiples parámetros de búsqueda.

Ante la posibilidad de que el uso de *Derived Query Methods* no satisfaga las necesidades del proyecto, se pueden implementar métodos personalizados. Sin embargo, dichos métodos continúan siendo más sencillos de implementar que los equivalentes en JDBC.

El requerimiento esencial para su implementación es la creación de una interfaz y una clase que deberán seguir una denominación específica. Suponiendo que el repositorio principal se llama *MiClaseRepository*, la interfaz se denominará *MiClaseRepositoryCustom* y, por último, se creará *MiClaseRepositoryImpl*, que implementará la interfaz. *MiClaseRepository* ahora deberá extender *MiClaseRepositoryCustom*.

Los métodos que requieran de este recurso no serán responsables de establecer ningún tipo de conexión, como sucedía en el capítulo 4. Realizarán consultas a través del *EntityManager* de Java Persistence Api, de forma más sencilla que antes.

```
@Transactional
public void createOrders(RequestOrder requestOrder) {
 entityManager.joinTransaction();
 entityManager.createNativeQuery("INSERT INTO Orders
  (address, user_id) VALUES (?,?)")
        .setParameter(1, requestOrder.getAddress())
        .setParameter(2, requestOrder.getUserId())
        .executeUpdate();
}
```

5.2. Desarrollo

Dado que ya se han mostrado las diferencias en los paquetes *domain* y *persistence*, este apartado se dedicará a hacer lo propio con los paquetes restantes, es decir, *controller* y *service*.

La modificación principal en el *controller* es la adición del método para registrar nuevos clientes. El *controller* recibe todos los datos necesarios para crear un usuario, excepto el valor de `user_id` que se genera de forma automática en la base de datos. Dichos datos se reciben encapsulados en un objeto denominado *RequestSignUp*. Tras recibirlos, el *cotroller* los envía al *service*, que realiza la llamada al método `createUser(RequestSignUp request)`. Este método es de estructura muy similar al que se muestra en las líneas de código anteriores, puesto que ambos realizan *inserts* en la base de datos.

En cuanto al *service*, ahora se instancia cada uno de los repositorios en *ElectronicsStoreServiceImpl*, y a través de dichas instancias se llama a cada uno de los métodos necesarios para realizar todas las tareas de la aplicación.

Capítulo 6

Aplicación de microservicios final

En el presente capítulo se narra el desarrollo de un nuevo microservicio, que se añade a la funcionalidad obtenida en el capítulo anterior.

Con esta incorporación en la arquitectura del sistema se pretende reiterar la facilidad de escalamiento de los microservicios, además de ilustrar el proceso para lograr la adición de un microservicio.

6.1. Microservicio *Points*

El nuevo microservicio tendrá una funcionalidad menos completa que el anterior, puesto que su principal objetivo no es la ampliación de la funcionalidad sino ilustrar cómo añadir un nuevo microservicio.

Los objetivos más importantes al crear un nuevo microservicio serán:

- Separación de conceptos: se debe mantener independiente del primer microservicio. Dado que realizará tareas que no van en la misma dirección que las del otro microservicio, se toma la decisión de separarlo. De lo contrario, se estaría dando un enfoque monolítico al asumir que el crecimiento de la aplicación supone ampliar la parte de la aplicación ya existente, en lugar de añadir nuevas partes.
- Acoplamiento débil: está relacionado con el anterior. Se deben mantener aislados los métodos de un microservicio de los métodos del otros.
- Escalabilidad: se desea mantener la facilidad de crecimiento de la aplicación.

El objetivo de este nuevo microservicio es otorgar puntos a los clientes en función de sus compras, para posteriormente conseguir descuentos. Cada euro gastado por el cliente le proporcionará mil puntos.

Puesto que para realizar esa tarea necesitará conocer los pedidos de los usuarios, se debe estudiar cómo conectar este microservicio al anterior. Para hacerlo de forma que se cumplan los objetivos expuestos anteriormente, la opción más apropiada será que el microservicio *store* avise al microservicio *points* cuando se formalice un pedido, y le envíe la información necesaria. Este enfoque se conoce como *event-driven* o dirigido por eventos. Por tanto, la realización de un pedido será un evento del que *store* avisará para que las partes interesadas (en este caso, el nuevo microservicio) puedan procesar esa información y realizar sus tareas.

6.2. Desarrollo

Se deberán incorporar nuevas tecnologías para poder trabajar con eventos. La primera de ellas será Spring AMQP - *Advanced Message Queuing Protocol* o Protocolo Avanzado de Distribución de Mensajes - que se comporta como un intermediario entre dos entidades que necesitan comunicarse.

También se utilizará RabbitMQ, que es un *software* de negociación de mensajes de código abierto. Además, implementa el protocolo AMQP mencionado anteriormente.

El primer paso será incluir la dependencia `spring-boot-starter-amqp` en el archivo `pom.xml`, que permite el uso de ambas tecnologías.

A continuación, se crea una clase de la que Spring podrá tomar las indicaciones sobre cómo configurar la comunicación. En un nuevo paquete denominado `configuration` se añade el archivo `RabbitMQConfiguration.java`, que contendrá la anotación @Configuration de Spring.

Asimismo, se crea el evento que se ha descrito en la sección anterior. Este se corresponde con la nueva clase `OrderCreatedEvent.java`, también ubicada en el paquete *configuration*.

Por último, se realiza el envío del evento. Para ello, se llama al método `send()` desde `ElectronicsStoreServiceImpl.java` tras la llamada al método que inserta nuevos pedidos en la tabla *orders*:

```
//communicates order created via event
 eventDispatcher.send(new OrderCreatedEvent(order_number,
    user.getUserId(), points));
```

Tras crear un nuevo proyecto Spring, como se explicó en capítulos anteriores, se crea un nuevo paquete `configuration`, tal y como se ha hecho en el microservicio existente. Se deberá crear también otro archivo `RabbitMQConfiguration.java`, que se enlazará con el ya existente.

Tras ello, se creará un *EventHandler*, que recibe el evento creado en el otro lado de la comunicación.

En el dominio, añadiremos copias de `Orders.java` y `User.java`.

En el *controller* se implementa un solo método. Será un GET que recibe el número de pedido.

Por último, se le indica a *points* cómo encontrar *store*. Para ello se edita el archivo `application.properties`:

```
#REST client configuration
storeHost = http://localhost:8080
```

Capítulo 7

Resultados y conclusiones

En el presente capítulo se discutirán los resultados obtenidos a lo largo del desarrollo del sistema, y se presentarán algunas conclusiones sobre los mismos.

Puesto que el desarrollo ha sido gradual, para evitar repeticiones se mostrarán exclusivamente capturas de la última aplicación, ya que es la más completa y de mayor complejidad y ayuda a obtener una visión más global de los resultados.

Así, se mostrarán uno por uno los resultados de las funciones que esta aplicación implementa. En primer lugar, se crea un cliente nuevo con el método `signUp()`. Se añaden los datos de este cliente en Postman y se envía la petición:

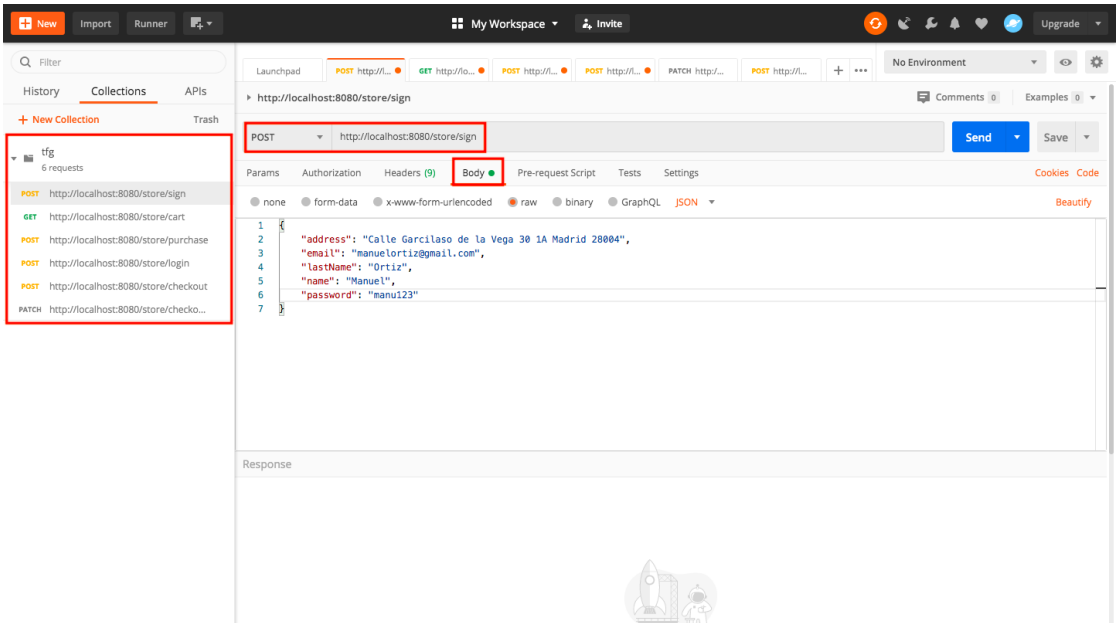


Figura 7.1: Creación de un cliente en el sistema desde Postman.

Se comprueba que este nuevo cliente se ha añadido a la base de datos:

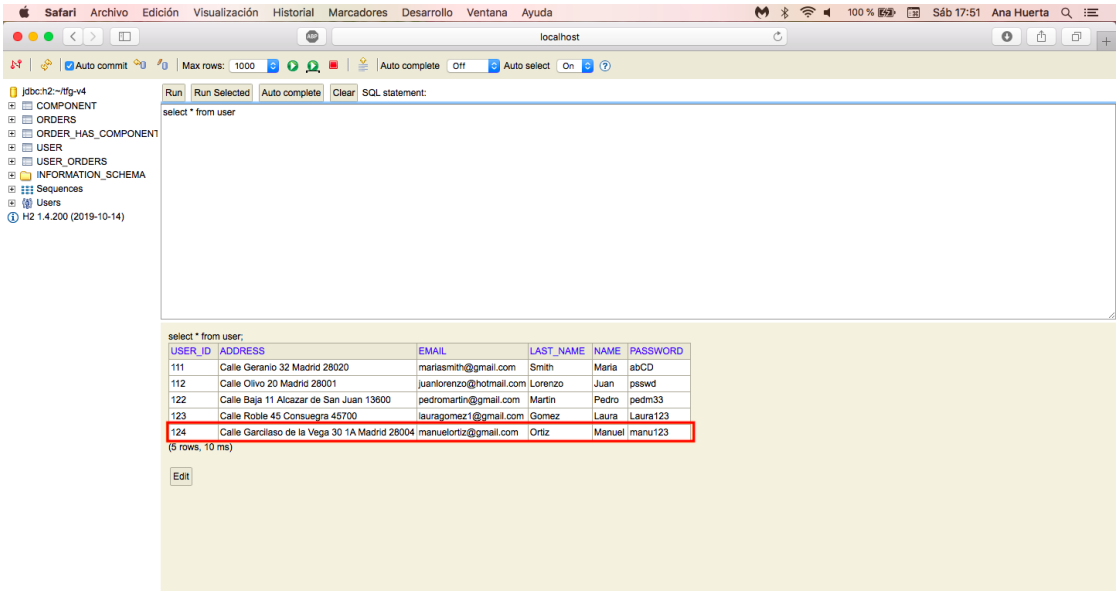


Figura 7.2: Resultado de la creación de un cliente.

El siguiente método que evaluar es `login()`. Se introducen en Postman los credenciales de un cliente ya existente en la base de datos y se envía la petición. El resultado es satisfactorio.

El siguiente paso será añadir artículos al carro de compra. Para lograrlo, se envían las referencias de dichos artículos desde Postman. Posteriormente, se comprueba que efectivamente aparecen en el carro de compra.

Ahora se procede a confirmar la compra. Esto requiere de dos pasos. En primer lugar, el método `checkout()` mostrará la lista de productos que se van a comprar. Finalmente, el método `confirmation()` creará el pedido, con todo lo que esto conlleva. Es decir, añadirá un registro en la tabla *orders*, varios en la tabla *order_has_component* (tantos como productos se compren), y otro en la tabla *user_orders* y finalmente, modificará las existencias de los productos correspondientes.

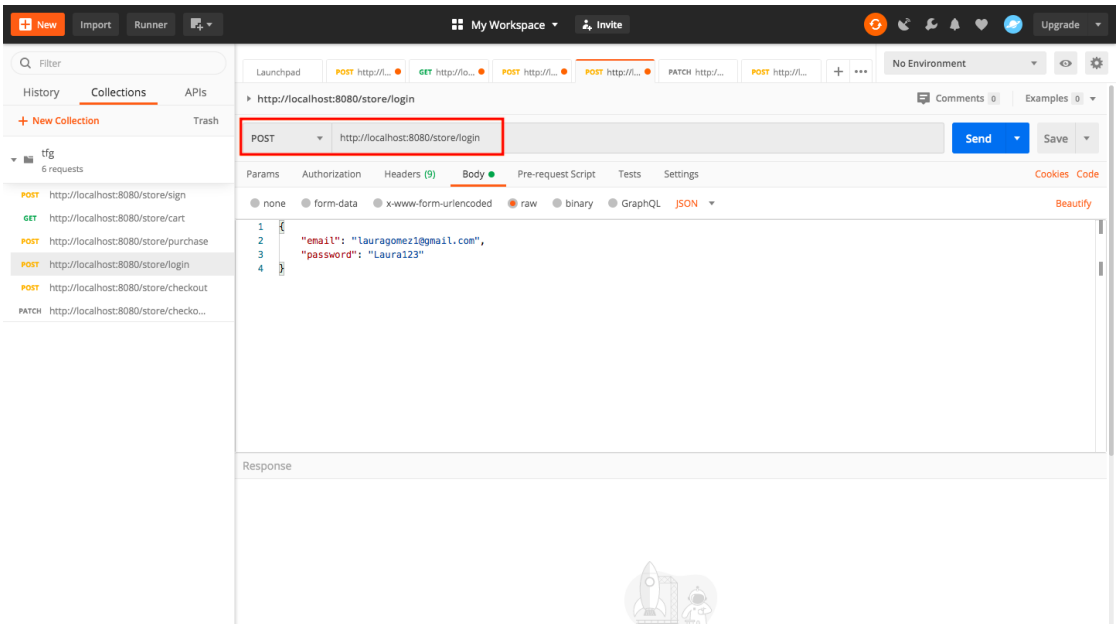


Figura 7.3: Acceso de un cliente al sistema desde Postman.

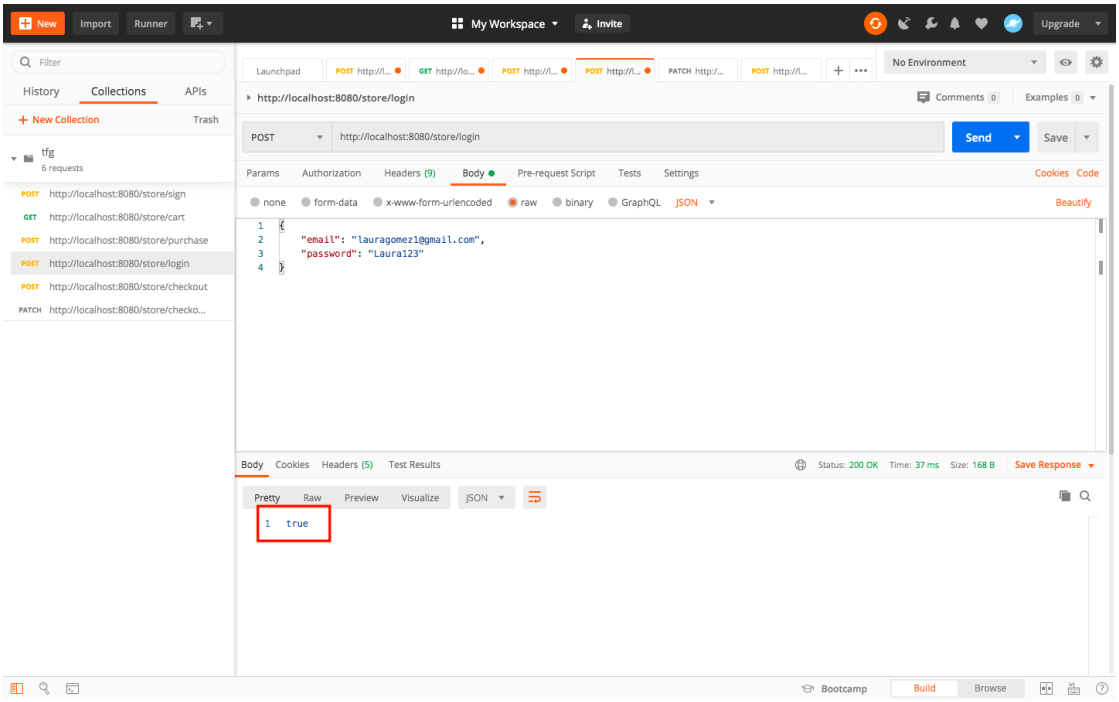


Figura 7.4: Resultado del acceso de un cliente al sistema desde Postman.

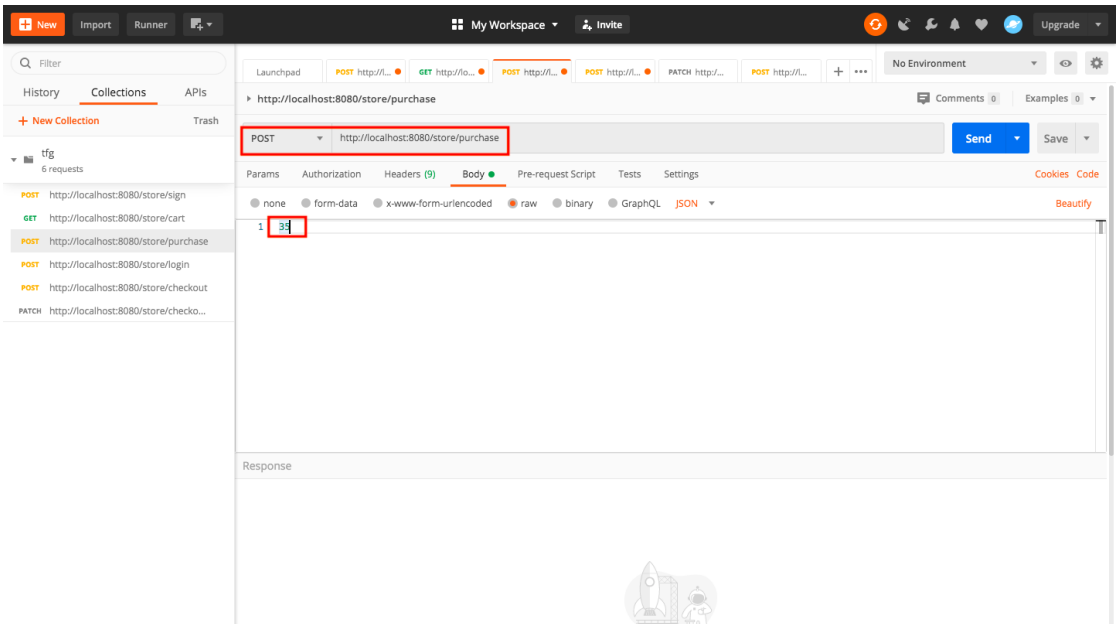


Figura 7.5: Se añaden artículos al carro de compra desde Postman.

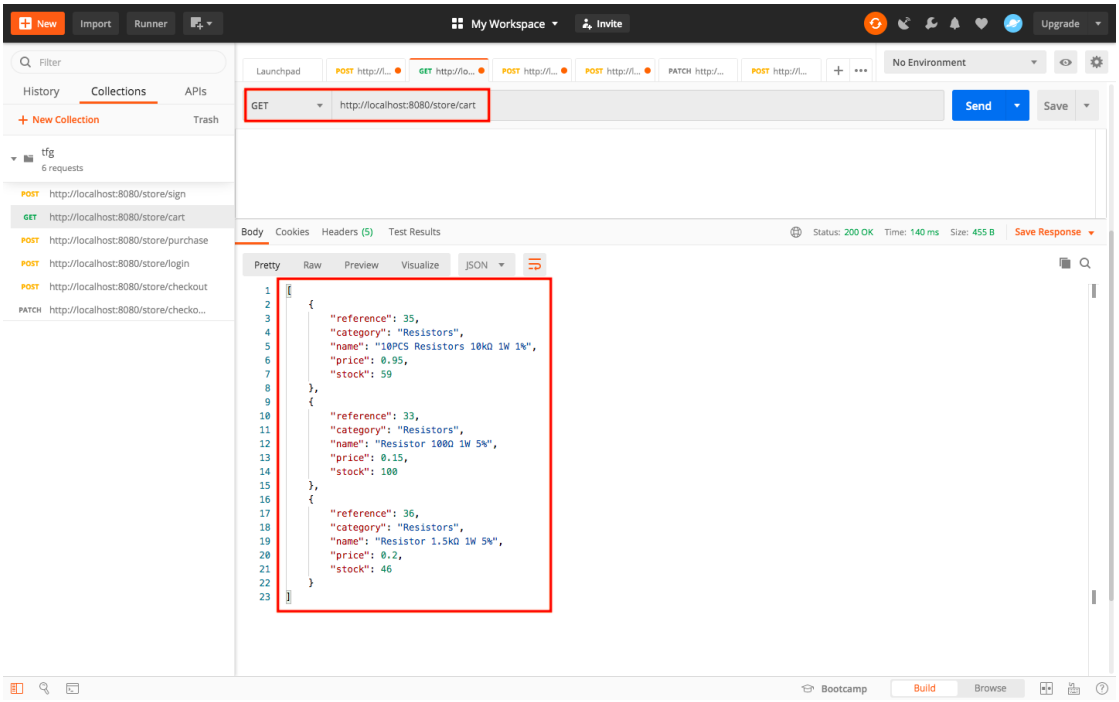


Figura 7.6: Resultado tras añadir tres artículos.

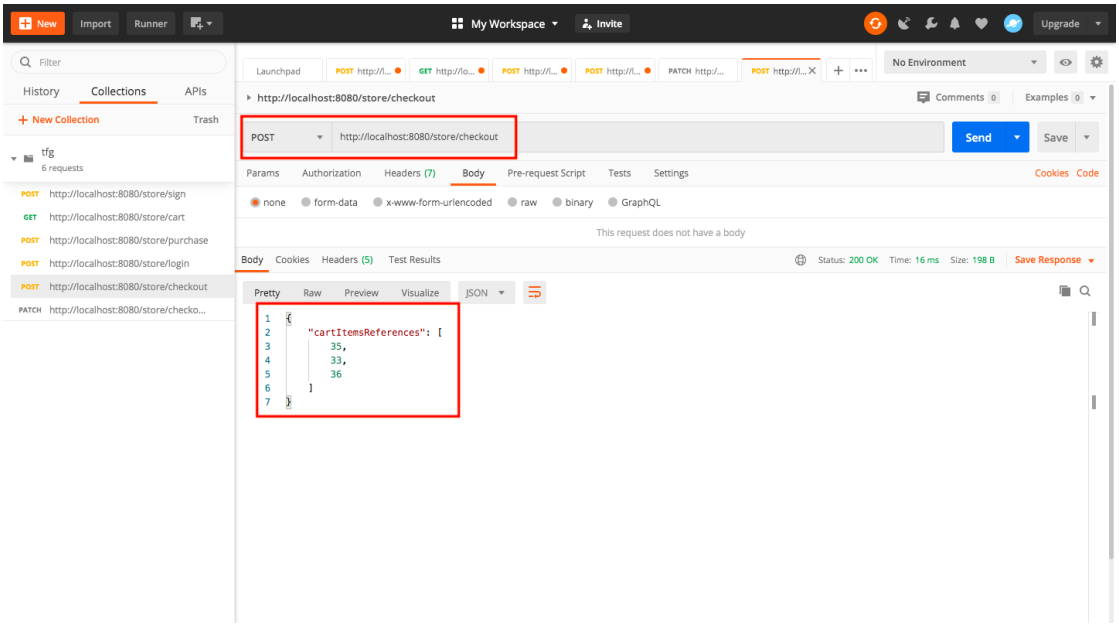


Figura 7.7: Se muestran los artículos del carro de compra antes de confirmar.

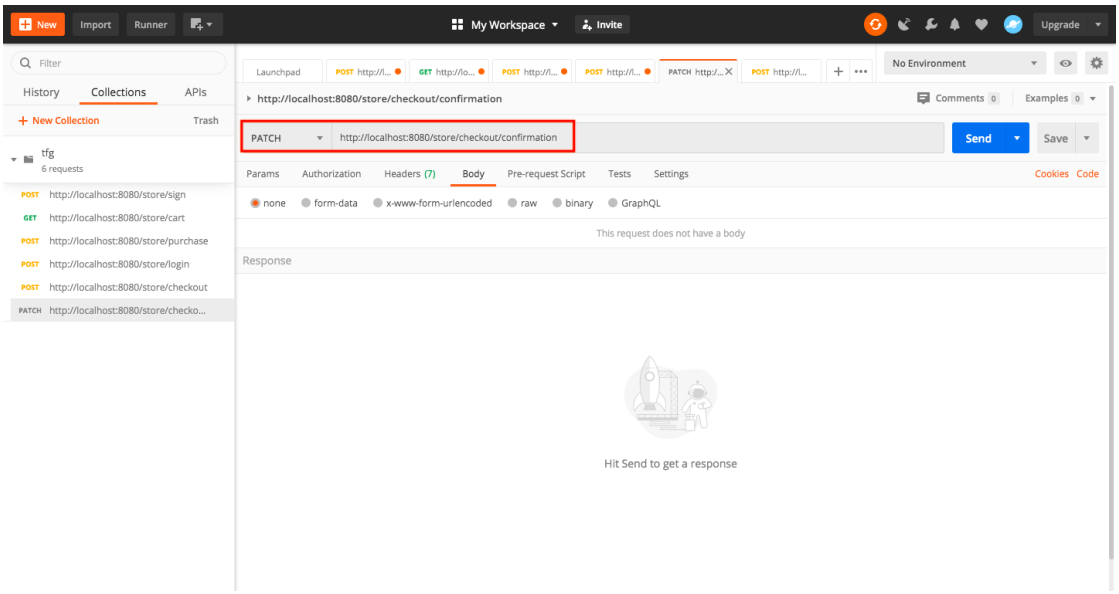


Figura 7.8: Se confirma el pedido.

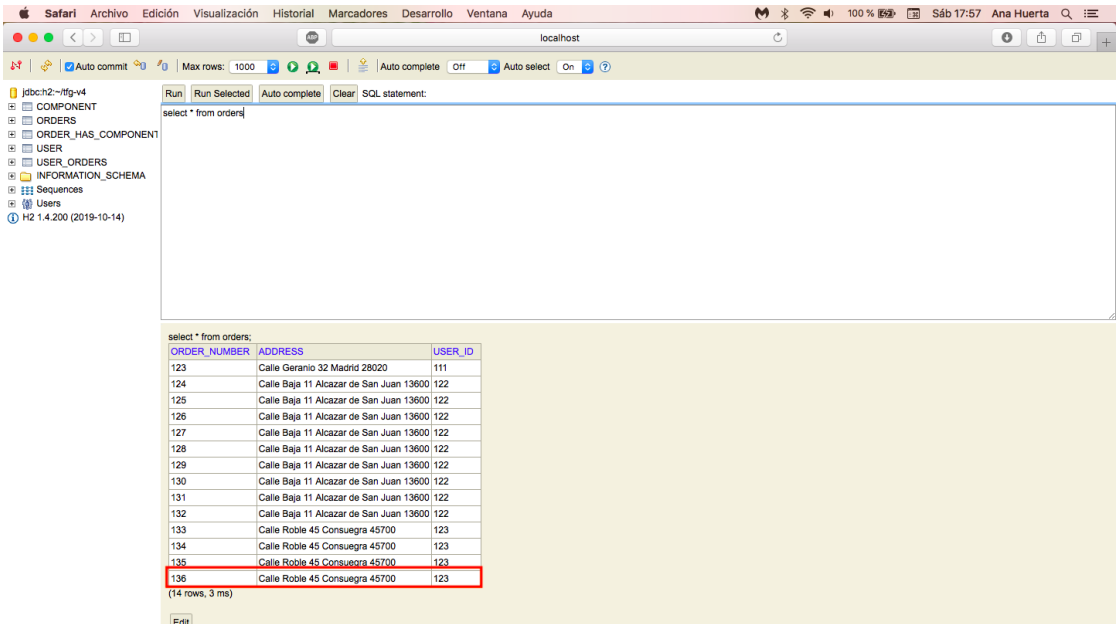


Figura 7.9: Nuevo registro en la tabla *orders*.

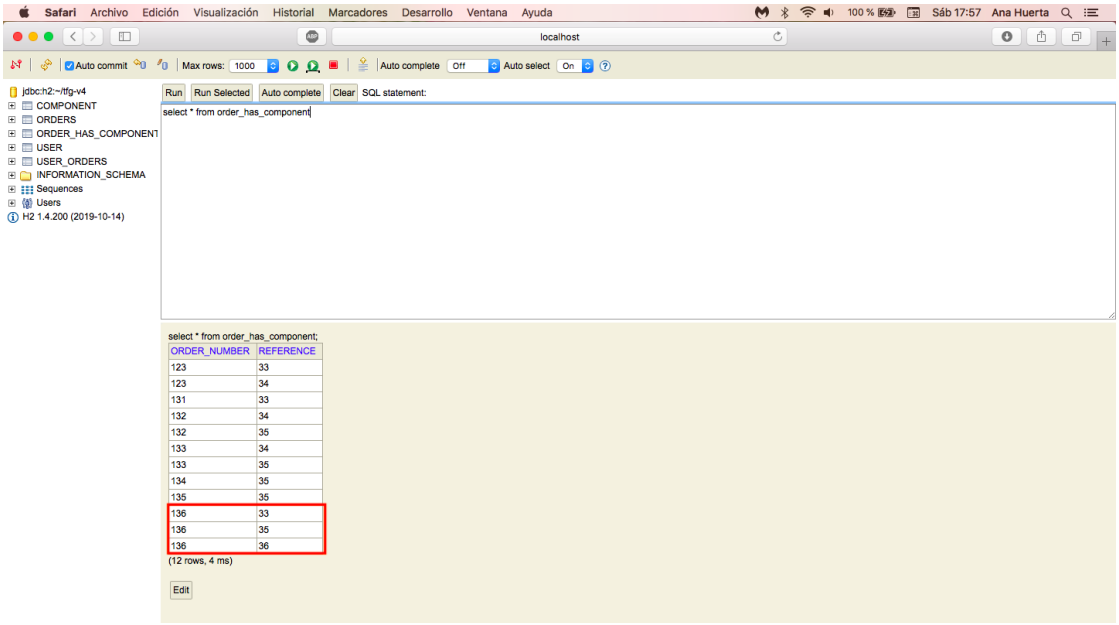


Figura 7.10: Nuevos registros en la tabla *order_has_component*.

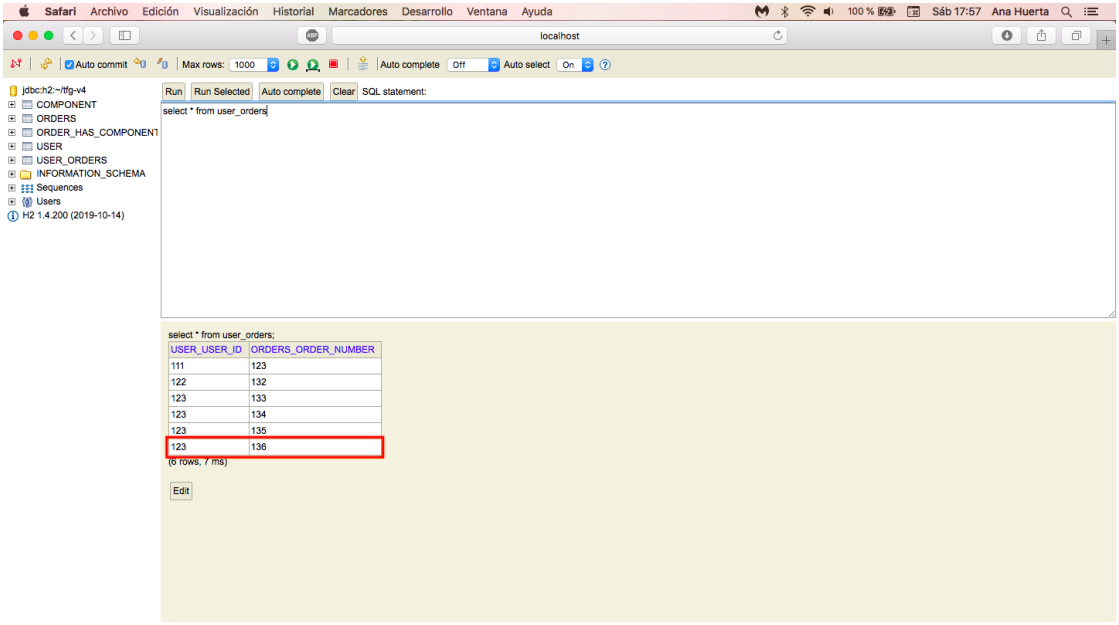


Figura 7.11: Nuevo registro en la tabla *user_orders*.

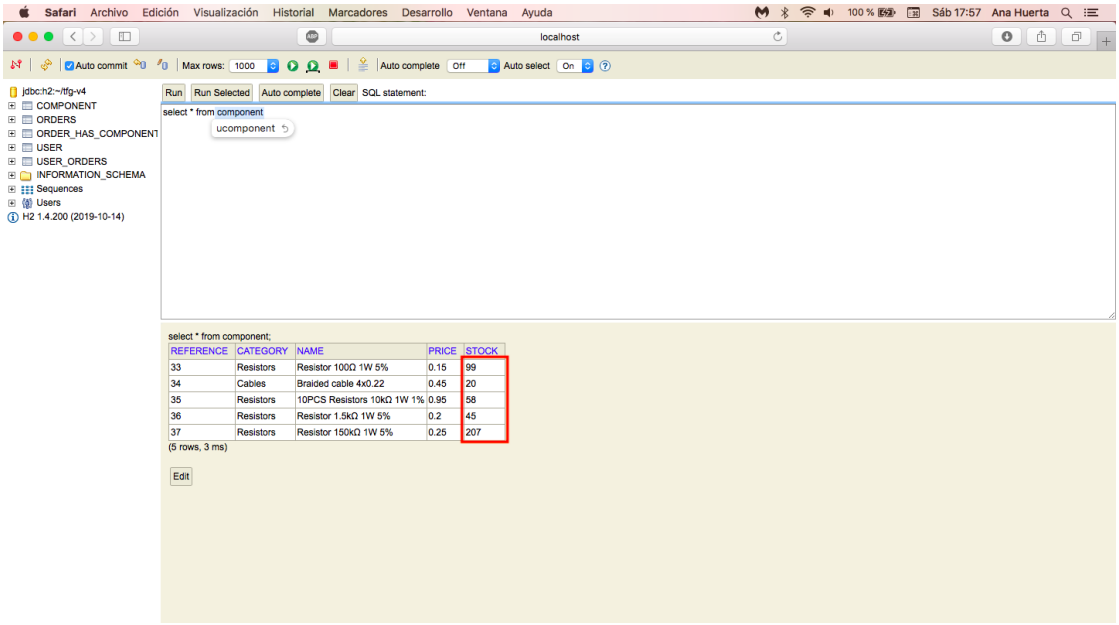


Figura 7.12: Modificación de las existencias de los productos.

Se ha diseñado una aplicación con arquitectura de microservicios de una manera gradual, y comparándola a lo largo del proceso con una aplicación monolítica. Con el objetivo de hacer más visibles esas comparaciones, se ha diseñado a su vez una aplicación monolítica, también de manera gradual.

Para todo ello, se ha empleado un número de tecnologías muy actuales, y algunas de ellas de importante complejidad.

Los resultados muestran el correcto funcionamiento de cada una de las funcionalidades desarrolladas en el proyecto.

Se puede concluir, por tanto, que el proyecto cumple satisfactoriamente los objetivos planteados en su propuesta. Puesto que siempre hay posibilidad de mejorar, el siguiente capítulo enumerará una serie de posibles desarrollos futuros.

Capítulo 8

Futuros Desarrollos

El proyecto introducido en este documento permite numerosas opciones de desarrollos futuros. Algunas de ellas son funcionalidades esenciales para hacer más realista la aplicación desarrollada, pero no se han implementado por las limitaciones de tiempo que brinda la asignatura. Estas posibles mejoras son:

- La compra de artículos no cuenta con ningún tipo de procesamiento de métodos de pago. Esta funcionalidad sería necesaria para una tienda *online* real, e interesante de desarrollar por los retos que plantea.
- La base de datos no implementa seguridad para los clientes de la tienda registrados en la plataforma.
- El evento creado en el segundo microservicio podría utilizarse para ampliar la aplicación con nueva funcionalidad. También podrían crearse eventos distintos dado que la lógica de envío ya está implementada, facilitando esta tarea.

Apéndice A

Alineación del proyecto con los Objetivos de Desarrollo Sostenible de la ONU

Tras estudiar los objetivos de desarrollo sostenible, se determina que el proyecto está más alineado con el noveno objetivo.

Dicho objetivo pretende construir infraestructuras resilientes y consolidar una industria sostenible e innovadora. Con ello, se podrían generar numerosos empleos e ingresos. Además, la industria desempeña un papel esencial en la introducción y desarrollo de nuevas tecnologías, el fomento del comercio internacional y la promoción del uso eficiente de recursos.

Sin embargo, la Organización de las Naciones Unidas considera que el mundo aún se encuentra lejos de alcanzar la cúspide de dicho potencial. El crecimiento del sector manufacturero ha ido disminuyendo constantemente en todo el mundo. Preocupan especialmente los países menos desarrollados, cuyos sectores manufactureros están lejos de la meta de 2030.

Estos datos son opuestos a las metas que se quieren alcanzar. La innovación tecnológica es decisiva para implementar soluciones duraderas a los desafíos económicos y medioambientales, lo que convierte la consecución de otros objetivos de desarrollo sostenible en dependientes de este [22].

Puesto que el objetivo de este proyecto es el estudio y divulgación de la transición a una arquitectura de microservicios, comparte las metas del objetivo de desarrollo sostenible introducido en las líneas anteriores.

La transformación de sistemas monolíticos en nuevos sistemas con arquitectura de microservicios proporciona una mejora a diferentes niveles, pero especialmente

Figura A.1: Listado de Objetivos de Desarrollo Sostenible.

en eficiencia. Este diseño admite que cada una de sus funciones responda de manera individual y no actúe como un todo, lo cual permite su reutilización para diferentes aplicaciones y, por tanto, da lugar a un uso eficiente de los recursos. También permite que estos recursos sean más fáciles de mantener, evitando que se disparen los costos.

El objetivo de desarrollo sostenible se divide en ocho metas que aseguren su consecución. Entre todas ellas, se aprecia que las metas 9.3 y 9.4 guardan estrecha relación con este proyecto.

La meta 9.3 establece la necesidad de aumentar la integración en los mercados de pequeñas empresas de la industria, así como su acceso a servicios financieros.

La PSD2 es una regulación europea reciente que está relacionada con la meta 9.3, puesto que intenta promover la innovación a través de la adaptación de los servicios bancarios a las nuevas tecnologías. Esta legislación, que entró en vigor en enero de 2018, concede acceso a terceros a la infraestructura de los bancos [23]. Así, las APIs - o Application Program Interface - que grandes empresas financieras están haciendo públicas son en la actualidad, en su mayoría, arquitecturas de microservicios.

Esto pone en evidencia la importancia de la divulgación de información educativa sobre microservicios, que permite a pequeñas empresas de la industria mejorar su infraestructura y, por tanto, su competitividad al equiparse de las últimas tecnologías, para así poder entrar al mercado en igualdad de condiciones que otras grandes empresas.

En la siguiente meta, la 9.4, también se manifiestan ideas afines al proyecto. Dicha meta propone transformar la industria en los próximos diez años. Se desea lograr que las industrias sean sostenibles, utilicen los recursos de manera eficiente y empleen procesos ambientalmente limpios.

Este proyecto pretende alentar la transición a la arquitectura de microservicios, que presentan mayor eficiencia que los monolitos. Sin embargo, las arquitecturas monolíticas aún están presentes en numerosas empresas. Según un estudio reciente, el 63 % de empresas entrevistadas están utilizando microservicios en algunas (18 %) o todas (46 %) sus aplicaciones. Del 37 % restante, un 28 % se plantea incorporarlos en un futuro cercano. Según estos datos, solo 9 de cada 100 empresas desea mantener su arquitectura monolítica. Esto implicaría que un gran número de empresas necesitan información fiable sobre cómo transformar sus aplicaciones [24].

Otro estudio relevante publica que un 38.7 % de los encuestados no utiliza arquitecturas de microservicios, pero se plantea hacerlo, y un 11.6 % no los utiliza y no considera hacerlo. Posteriormente, la pregunta sobre por qué no la emplean muestra resultados muy relevantes para el proyecto:

- 24.4 % por falta de aprendizaje de la materia.
- 6.7 % por falta de tiempo para enseñar a los desarrolladores.
- 22.2 % por falta de tiempo para refactorizar sus aplicaciones.
- 37.8 % por falta de conocimiento sobre el tema [25].

Este proyecto espera poder convertirse en una de las guías que posibiliten y faciliten dicha transición, para así contribuir a la modernización de la industria y a la búsqueda de tecnologías más eficientes.

Bibliografía

[1] R. L. Grossman. *The Structure of Digital Computing. From Mainframes to Big Data.* URL: http://www.structureofdigitalcomputing.com/structure-chapter-01.pdf.

[2] M. Anjarwala. *How Software Development has changed in the Past Decade.* Blog Entry. 2019. URL: https://xcelacore.com/how-software-development-has-changed-in-the-past-decade/.

[3] M. Rouse. *Definition of monolithic architecture.* 2016. URL: https://whatis.techtarget.com/definition/monolithic-architecture.

[4] R. Jin. *Microservices: Terribly Named, Ambiguously Defined.* 2017. URL: https://thenewstack.io/microservices-terribly-named-ambiguously-defined/.

[5] K. Brown. *Beyond buzzwords: A brief history of microservices patterns.* Article. 2018. URL: https://developer.ibm.com/technologies/microservices/articles/cl-evolution-microservices-patterns/.

[6] *Tecnología web para desarrolladores: HTTP.* URL: https://developer.mozilla.org/es/docs/Web/HTTP.

[7] BBVA Open4U. *API REST: qué es y cuáles son sus ventajas en el desarrollo de proyectos.* URL: https://bbvaopen4u.com/es/actualidad/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos.

[8] S. Mane. *Understanding REST.* Article. 2017. URL: https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa.

[9] *URI: ¿qué es el identificador de recursos uniforme?* 2020. URL: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/uri-identificador-de-recursos-uniformes/.

[10] *Qué son microservicios y ejemplos reales de uso.* 2016. URL: https://openwebinars.net/blog/microservicios-que-son/.

[11] *Estilo de arquitectura de microservicios*. 2019. URL: https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices.

[12] *Eclipse IDE*. 2014. URL: https://www.genbeta.com/desarrollo/eclipse-ide.

[13] Y. Muradas. *Spring Framework*. 2018. URL: https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/.

[14] *git - About*. URL: https://git-scm.com/about.

[15] *Entiende qué es Maven en menos de 10 min*. URL: https://www.javiergarzas.com/2014/06/maven-en-10-min.html.

[16] *¿Qué es MySQL?* URL: https://www.hostinger.es/tutoriales/que-es-mysql/.

[17] F.J. Martínez Páez. *Introducción a JDBC*. 2006. URL: https://www.adictosaltrabajo.com/2006/05/04/introjdbc/.

[18] O. Blancarte. *Java Persistence API JPA*. URL: https://www.oscarblancarteblog.com/tutoriales/java-persistence-api-jpa/.

[19] Equipo Geek Everis. *¿Qué es Java Hibernate? ¿Por qué usarlo?* 2019. URL: https://ifgeekthen.everis.com/es/que-es-java-hibernate-por-que-usarlo.

[20] *H2 Database Tutorial*. URL: https://www.tutorialspoint.com/h2_database/index.htm.

[21] F. Redondo. *Postman: gestiona y construye tus APIs rápidamente*. 2017. URL: https://www.paradigmadigital.com/dev/postman-gestiona-construye-tus-apis-rapidamente/.

[22] *Objetivo 9: Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación*. URL: https://www.un.org/sustainabledevelopment/es/infrastructure/.

[23] *Todo lo que hay que saber de la PSD2*. URL: https://www.bbva.com/es/lo-saber-la-psd2/.

[24] *New research shows 63 % of enterprises are adopting microservices architectures*. URL: https://dzone.com/articles/new-research-shows-63-percent-of-enterprises-are-a.

[25] *DZone Research: Microservices Priorities and Trends*. URL: https://dzone.com/articles/dzone-research-microservices-priorities-and-trends.